MULTI-2D Report

MAX-PLANCK-INSTITUT FÜR QUANTENOPTIK

MULTI2D - A Computer Code for Two-Dimensional Radiation Hydrodynamics

R. Ramis and J. Meyer-ter-Vehn

Max-Planck-Institut für Quantenoptik, 8046 Garching, Fed. Rep. Germany

MPQ 174

August 1992

MULTI2D - A COMPUTER CODE FOR TWO-DIMENSIONAL RADIATION HYDRODYNAMICS

R. Ramis* and J. Meyer-ter-Vehn

Max-Planck-Institut für Quantenoptik, 8046 Garching, Fed. Rep. Germany

August 1992

Abstract: The interpretation of high-power laser experiments at MPQ requires numerical simulation of radiation hydrodynamics in two and three spatial dimensions. In the experiments, high-Z target configurations of different geometries are heated by the laser pulse, and the energy deposited in the laser spot is transported away predominantly by soft x-rays. The difficulty is that radiation transport proceeds through expanding plasma which is optically thick in some regions and optically thin in others. A diffusion description is inadequate in this case. A new numerical code has been developed which describes hydrodynamics in two spatial dimensions (cylindrical r, z geometry) and radiation transport along rays in three dimensions with the 4π solid angle discretized in 64 directions. Matter moves on a non-structured, triangular mesh. Radiation is transported according to a novel scheme; radiation flux of a given direction enters on two (one) sides of a triangle and leaves on the opposite side(s) in proportion to the viewing angles depending on the geometry. This scheme allows to propagate sharply edged beams without ray tracing, though at the price of some lateral diffusion.

* On leave from Escuela Técnica Superior de Ingenieros Aeronáuticos, Universidad Politécnica de Madrid, 28040 Madrid, Spain.

Table of content

1. Physical and Numerical Model

1.1 Radiation transport equations

In ICF targets and related laser-plasma experiments, the velocity of matter is usually much smaller than the speed of light. The radiation field can be regarded as quasi-steady at any instant of time. The spectral radiation intensity $I(\vec{n}, \vec{r}, \nu)$ is determined (neglecting scattering) by the transport equation (Zel'dovich 2.34)

$$\vec{n} \cdot \nabla I = \frac{I_E - I}{\lambda}$$

where λ is the mean free path of radiation, depending on the frequency and the material density and temperature, and I_E is the planckian equilibrium intensity given by the matter temperature (Zel'dovich 2.11) according to

$$I_E = \frac{2h\nu^3}{c^2} \frac{1}{e^{h\nu/kT} - 1}.$$

The values of the physical constants are $c = 2.998 \cdot 10^{10} cm/sec$, $h = 6.625 \cdot 10^{-27} erg \cdot sec$, and $k = 1.602 \cdot 10^{-12} erg/eV$. Once the radiation field is known, the energy gained per unit volume of matter is given by the integral over the entire solid angle and over all frequencies (Zel'dovich 2.55)

$$Q = \int \frac{I - I_E}{\lambda} \, d\vec{n} \, d\nu.$$

In the case that λ does not depend on frequency, one can intregate the above equations over all frequencies. The integrated radiation intensity $I'(\vec{n}, \vec{r}) = \int I \, d\nu$ is determined by

$$\vec{n} \cdot \nabla I' = \frac{I'_E - I'}{\lambda}$$

and the energy gained by the matter by

$$Q = \int \frac{I' - I'_E}{\lambda} \, d\vec{n},$$

where I'_E is the equilibrium intensity given by (Zel'dovich 2.14)

$$I'_E = \int I_E \, d\nu \simeq \frac{\sigma T^4}{\pi}$$

$$\sigma = \frac{2\pi^5 k^4}{15 h^3 c^2} = 1.03 \cdot 10^{12} erg/cm^2 \cdot sec \cdot ev^4.$$

These expressions can also be used in the more general case of λ depending on the frequency, assuming some appropriate mean value for λ. This is the so called "Gray Approximation". Average values commonly used for the mean free path are the Planck mean (Zel'dovich 2.105) and Rosseland mean (Zel'dovich 2.77), appropriate for optically thin ($\lambda \gg L$), and optically thick ($\lambda \ll L$) regimes, respectivelly (L being a typical length of the problem).

In a multigroup treatment, on the other hand, one integrates over N intervals in frequency, obtaining N equations, each similar to te one above, but with a different meaning of I'_E. For simplicity, we will consider only the gray case (the primes will dropped in the following development). This is what is implemented currently (July 1992) in the code. Typical values of λ are summarized in the appendix.

1.2 Discretization in direction

We consider only axisymmetrical configurations. In cylindrical coordinates (z, r, and ψ), all quantities are independent of ψ. The unit vector in the direction of radiation propagation is written as

$$\vec{n} = \cos\theta\vec{u}_z + \sin\theta\cos\phi\vec{u}_r + \sin\theta\sin\phi\vec{u}_\psi.$$

The transfer equation for the intensity $I = I(r, z, \theta, \phi)$ is now

$$\sin\theta(\cos\phi\frac{\partial I}{\partial r} - \frac{\sin\phi}{r}\frac{\partial I}{\partial\phi}) + \cos\theta\frac{\partial I}{\partial z} = \frac{I_E - I}{\lambda},$$

whereas the energy deposited into matter becomes

$$Q = \int_{-\pi}^{\pi}\int_{0}^{\pi}\frac{I - I_E}{\lambda}\sin\theta\, d\theta\, d\phi.$$

We assume also mirror symmetry for each plane containing the Z-axis; $I(\phi) = I(-\phi)$. The planar case can be considered as a limit of the axisymmetrical case when $r \to \infty$, and will be not discussed separately. The radiation field will be first discretized in direction. The unit sphere is divided into a finite number of zones (beamlets), and a uniform value of intensity is assumed in each. For clarity, this procedure will be done in two steps. First we subdivide the range of variation of θ into M intervals, assuming the intensity to be a function of ϕ only, in each of them. $I(r, z, \theta, \phi) = I_i(r, z, \phi)$ for $\theta_i < \theta < \theta_{i+1}$, being $\theta_1 = 0$, and $\theta_{M+1} = \pi$. Multiplying the transfer equation by $\sin\theta\, d\theta$, and integrating over each interval (lowest order moment, in order to conserve the number of photons), one arrives at a set of equations in the form

$$\alpha_i(\cos\phi\frac{\partial I_i}{\partial r} - \frac{\sin\phi}{r}\frac{\partial I_i}{\partial\phi}) + \beta_i\frac{\partial I_i}{\partial z} = \gamma_i\frac{I_E - I_i}{\lambda}$$

with the coefficients

$$\alpha_i = \frac{\theta_{i+1} - \theta_i}{2} + \frac{\sin 2\theta_i - \sin 2\theta_{i+1}}{4}$$

$$\beta_i = \frac{\cos 2\theta_i - \cos 2\theta_{i+1}}{4}$$

$$\gamma_i = \cos\theta_i - \cos\theta_{i+1}$$

The deposition of radiation into matter becomes

$$Q = \sum_i\int_{-\pi}^{\pi}\gamma_i\frac{I_i - I_E}{\lambda}d\phi.$$

In a second step, we subdivide one half of the range of variation of ϕ (the other half is symmetric as pointed out above) into N_i intervals (this number depending on i in general), and assume the radiation to have constant intensity in each: $I_i(r, z, \phi) = I_{ij}(r, z)$ for $\phi_{ij} < \phi < \phi_{ij+1}$ being $\phi_{i1} = -\pi$,and $\phi_{iN_i+1} = 0$. Integrating the above equations one arrives at

$$A_{ij}\frac{\partial I_{ij}}{\partial r} + B_{ij-1}\frac{I_{ij} - I_{ij-1}}{r} + C_{ij}\frac{\partial I_{ij}}{\partial z} = D_{ij}\frac{I_E - I_{ij}}{\lambda}$$

with

$$A_{ij} = \alpha_i(\sin\phi_{ij+1} - \sin\phi_{ij})$$

$$B_{ij} = -\alpha_i\sin\phi_{ij+1}$$

$$C_{ij} = \beta_i(\phi_{ij+1} - \phi_{ij})$$

$$D_{ij} = \gamma_i(\phi_{ij+1} - \phi_{ij}).$$

Observe that the term $\partial I_i/\partial\phi$ is singular at ϕ_j because I_i is assumed discontinuous there. Although its integral can be computed there, it is not clear how to distribute its value between two adjacent intervals. To solve this ambiguity, we use the following physical argument: this term takes into account that photons traveling in straight lines change from one beamlet to the other due to a geometrical effect in cylindrical coordinates. It is reasonable to assume that the term in equation for I_{ij} depends on the intensity I_{ij-1} of the beamlet from which the photons are coming, but not on the intensity I_{ij+1} of the beamlet to which they are going. This assumption together with the fact that $I = I_E = Const.$ must be a solution, determines completely this term. The deposition of energy into matter can be written as

$$Q = \frac{2\sum_{i,j} D_{ij} I_{ij} - 4\pi I_E}{\lambda}.$$

The coefficients satisfy the following properties:

$$\sum_j A_{ij} = 0$$

$$\sum_i \sum_j C_{ij} = \sum_i \pi\beta_i = 0$$

$$\sum_i \sum_j D_{ij} = \sum_i \pi\gamma_i = 2\pi.$$

In order to get an equation in conservative form, suitable to be discretized in space, one can define a new variable $\Gamma_{ij} \equiv 2\pi r I_{ij}$. The equations now take the form

$$A_{ij}\frac{\partial\Gamma_{ij}}{\partial r} + \frac{B_{ij}\Gamma_{ij} - B_{ij-1}\Gamma_{ij-1}}{r} + C_{ij}\frac{\partial\Gamma_{ij}}{\partial z} = D_{ij}\frac{2\pi r I_E - \Gamma_{ij}}{\lambda}$$

$$2\pi r Q = \frac{2\sum_{i,j} D_{ij}\Gamma_{ij} - 8\pi^2 r I_E}{\lambda}.$$

The amount of energy deposited in a toroid, the section of which is A, is given by

$$\int_A 2\pi r Q\, dr\, dz = \sum_{i,j}\int_A -2D_{ij}\frac{2\pi r I_E - \Gamma_{ij}}{\lambda}\, dr\, dz = -\sum_{ij}\int_A \nabla\cdot\vec{S}_{ij}\, dr\, dz,$$

where the quantities $\vec{S}_{ij} = 2\left(A_{ij}\Gamma_{ij}\vec{u}_r + C_{ij}\Gamma_{ij}\vec{u}_z\right)$ are the energy fluxes (times $2\pi r$) carried on by each beamlet. Finally, we can define a new system of coordinates

$$l = z\cos\psi + r\sin\psi$$

$$s = -z\sin\psi + r\cos\psi.$$

Choosing $\tan\psi = A_{ij}/C_{ij}$, one arrives at equations in the form

$$E_{ij}\frac{\partial\Gamma_{ij}}{\partial l} + \frac{B_{ij}\Gamma_{ij} - B_{ij-1}\Gamma_{ij-1}}{r} = D_{ij}\frac{2\pi r I_E - \Gamma_{ij}}{\lambda}$$

$$\vec{S}_{ij} = 2E_{ij}\Gamma_{ij}\vec{u}_l$$

where

$$E_{ij} = \sqrt{A_{ij}^2 + C_{ij}^2}.$$

This defines the way in which the transport will be solved. One considers successively groups of beamlets with the same value of i. In each group one solves first the equation for Γ_{i1} along lines of

constant s. The natural boundary condition is the value of the incident flux on the matter at $l = l_{min}$. From this point one integrates the above equation up to $l = l_{max}$. The term $B_{ij}\Gamma_{ij}/r$ represents the transfer of photons to the next beamlet, which is zero in planar geometry (when $r \rightarrow \infty$). Once Γ_{i1} is known, one can solve a similar equation for Γ_{i2}, in which the term $B_{ij-1}\Gamma_{ij-1}/r$ represents the transfer of photons from the previous beamlet. A reflecting boundary condition can be easily implemented by first storing the values of the outgoing $\Gamma's$, and later using them in the boundary conditions for the incoming $\Gamma's$.

1.3 Discretization in space

To use the algorithm as described above, the separation between ray paths must be smaller than the minimum size of the mesh, in order to avoid numerical noise. But computational grids in typical radiation hydrodynamical problems can be very distorted with some of the cells having a very small thickness. The number of rays would be very large in such cases. We prefer to define the $\Gamma's$ on the mesh as any other quantity, so that the number of operations scales as the number of elements in the grid. We use a non-structured (ordered in an arbitrary way) grid formed by triangular elements. We define the values of $\Gamma's$ on the interfaces between elements. This is the natural place to define a flux. For a given beamlet, we compute the outgoing fluxes in each cell as a function of the incoming ones. This implies two cases: **i)** cells with one input and two outputs, **ii)** cells with two inputs and one output. Inside each cell, we approximate λ,Γ_{ij-1}, and r by constant values. The transport equation can be written as

$$E\frac{\partial\Gamma}{\partial l} + \frac{B\Gamma}{R} - T = D\frac{2\pi r I_E - \Gamma}{\lambda},$$

where $R = \langle r \rangle$, $T = \langle \Gamma_{ij-1}B_{ij-1}\rangle/R$, and the subindexes have been dropped. This equation can also be written as

$$\frac{\partial\Gamma}{\partial l} = \frac{\Gamma_E - \Gamma}{\lambda_E}$$

with $\lambda_E \equiv E/(D/\lambda + B/R)$, and $\Gamma_E \equiv (2\pi r I_E D/\lambda + T)/(D/\lambda + B/R)$. To apply this equation to a triangular cell, we assume first a mean value of Γ_E over each side, so that the source term Γ_E is a continuous function. Its continuity is required in order to recover the diffusion behaviour when $\lambda \rightarrow \infty$. In our numerical scheme, temperature and velocity are defined at the nodes (vertices of triangles), because they must be continuous in space, while density and material properties are defined at the cells centers, so that composite targets can be described properly. The value of λ is computed as a function of cell density, composition, and temperature (when required, a cell value of a node centered quantity is obtained as a mean value). The values of Γ_E are obtained with the mean values of r, and I_E ($\equiv \sigma T^4/\pi$), at the interfaces. Consider first a triangular cell of type i (Figure 1a). This can be divided into two triangular zones, each with only one input and one output (Figure 1b). Further, we can aproximate each of them, by a rectangle of the same area and height (Figure 1c). The problem is thus reduced to an one-dimensional one of the form:

$$\frac{\partial I}{\partial l} = \frac{S - I}{\lambda}, \qquad S = S_a + \frac{l}{L}(S_b - S_a), \qquad I(0) = I_a$$

with the solution

$$I(L) = (I_a - S_a)e^{-L/\lambda} + S_b - (1 - e^{-L/\lambda})\lambda\frac{S_b - S_a}{L}$$

$$\langle I \rangle \equiv \frac{\int_0^L I(x)\,dx}{L} = \frac{S_a + S_b}{2} - \frac{\lambda}{L}(I_b - I_a).$$

Applying these formulas to our problem, one arrives at

$$\Gamma^{CA} = (\Gamma^{BC} - \Gamma_E^{BC} + \frac{\lambda_E}{d}(\Gamma_E^{CA} - \Gamma_E^{BC}))\eta + \Gamma_E^{CA} - \frac{\lambda_E}{d}(\Gamma_E^{CA} - \Gamma_E^{BC})$$

$$\Gamma^{AB} = (\Gamma^{BC} - \Gamma_E^{BC} + \frac{\lambda_E}{d}(\Gamma_E^{AB} - \Gamma_E^{BC}))\eta + \Gamma_E^{AB} - \frac{\lambda_E}{d}(\Gamma_E^{AB} - \Gamma_E^{BC})$$

$$\langle\Gamma\rangle^{ABC} = f_c\langle\Gamma\rangle^{AA'C} + f_b\langle\Gamma\rangle^{ABA'} = \frac{f_c\Gamma_E^{CA} + f_b\Gamma_E^{AB} + \Gamma_E^{BC}}{2} - \frac{\lambda_E}{d}(f_c\Gamma^{CA} + f_b\Gamma^{AB} - \Gamma^{BC})$$

with $\eta = e^{-d/\lambda_E}$. In the case of a cell of type ii, one proceeds in a similar way, obtaining

$$\Gamma^{BC} = f_c\Gamma^{CA'} + f_b\Gamma^{A'B}$$

$$\Gamma^{BC} = (f_c(\Gamma^{CA} - \Gamma_E^{CA}) + f_b(\Gamma^{AB} - \Gamma_E^{AB}) + S_v)\eta + \Gamma_E^{BC} - S_v$$

$$\langle\Gamma\rangle^{ABC} = \frac{f_c\Gamma_E^{CA} + f_b\Gamma_E^{AB} + \Gamma_E^{BC}}{2} + \frac{\lambda_E}{d}(f_c\Gamma^{CA} + f_b\Gamma^{AB} - \Gamma^{BC})$$

with $S_v = \lambda_E(\Gamma_E^{BC} - f_c\Gamma_E^{CA} - f_b\Gamma_E^{AB})/d$. In this case, the two fluxes coming out of the two halfs of the cell have been mixed together. Although this will introduce some sort of numerical diffusion, its effect is small in comparison with other discretization effects, e.g. the finite number of directions or cells. Once the fluxes on the sides of a cell are known, the deposition of energy in the cell is given by (cell of type i):

$$2EH(\Gamma^{BC} - f_c\Gamma^{CA} - f_b\Gamma^{AB}).$$

The mean value $\langle\Gamma\rangle^{ABC}$ will be used to compute the value of T, i.e. the quantity of photons changing to the next beamlet inside of this cell. Once the dumped energy in each cell is known, it is necessary to distribute it to the nodes, where the energy equation has to be solved. For optically thin cells with positive deposition (the cell absorbs radiation), the energy is divided into three equal parts. If the deposition is negative (the cell emits radiation), the energy emitted by each node is assumed proportional to its temperature raised to the fourth power. This avoids to subtract energy from a node with small or zero temperature. For optically thick cells the algorithm is more sophisticate. Consider for the moment planar geometry. The basic idea is that inside each cell exists a quasi-equilibrium value of Γ given by

$$\Gamma_{eq} \simeq \Gamma_E - \lambda_E\frac{\partial\Gamma_E}{\partial l}.$$

Only at the places where Γ is different to this value, there is energy deposition or absorbtion. This can occur only at the entrance of the cells, because inside the cell Γ goes to Γ_E in a distance of the order of λ. Let us consider, for example, a cell of type i, in which the side BC is receiving some intensity Γ^{BC}. The energy deposited there (by this beamlet) would then be

$$HE(\Gamma^{BC} - \Gamma_{eq})$$

and would be distributed in equals parts between nodes B and C. Adding the contributions of the three sides of a cell for all beamlets, gives the same total as adding all the cell depositions in the form given above. (Terms containing the space derivative cancel when added for three sides, and on the other hand $\sum EH = \sum E(H_x\cos\psi - H_y\sin\psi) = H_x\sum C - H_y\sum A = 0$). The value Γ^{BC} is (except at the boundaries) very close to the quasi-equilibrium value in the previous cell. The dumped energy at the interface is thus proportional to the jump of the first derivative of Γ_E. This remains true when one adds the contributions of all beamlets: the deposited energy in each interface is the difference between the thermal flows of adjacent cells. This is equivalent to apply FEM (the finite element method) in the optically thick limit. For cylindrical geometry, things are just a bit more complicated. One has to use

$$\Gamma'_{eq} = \Gamma_{eq} + \frac{2\pi rI_E}{1 + DR/B\lambda}$$

to be consistent in the energy deposition.

Using this method, we have a correct description for both the optically thin and thick limit, and a smooth transition between them; the energy is strictely conserved, I is guaranted to be positive, and the boundary conditions are in general rather trivial (just specify incident I_E at boundary interfaces).

In practice, we subdivide the unit sphere into 32 or 64 beamlets. By symmetry only one half has to be solved.

1.4 Lagrangian Hydrodynamics

To solve the motion of the matter, we use a simple scheme that can be thought of as a straightforward extension to two dimensions of the lagrangian method described in the classical book of Richtmyer and Morton. We consider density, temperature, and pressure as cell centered quantities, and velocity, coordinates, and inertial mass as defined at the nodes (vertices of triangles). In developing the scheme for radiation transport in the previous section, we have assumed temperatures at the nodes. We have to deal with two diferent sets of temperatures. We consider the ones at the nodes as the primary ones, and we take mean values when we need values at the cells centers. Consequently, we will solve the energy equation at the nodes. Once we have temperatures and densities at time t, we can compute the pressure (at the moment only ideal gas equation of state is implemented in the code, so this process is very simple). We advance the configuration of the mesh, computing the new coordiantes by

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t.$$

New values of volume and density are then obtained. We take the expression $\pi(r_{min}+r_{max}) \times Area$ as the volume of a cell. It gives better results than taking the volume of the revolution toroid associated with the cell, in special near the symmetry axis. In order to stabilize the scheme in presence of shock waves, it is necessary to include some sort of dissipation in the form of an artificial viscous pressure. We use $q = a^2 \rho \sum \min(0, \partial l_k/\partial t)^2$, where a is a coefficient of order unity ($a = 2$ in our computations), and l_k are the characteristic lengths of the cell. We consider as such, the length of the longest side, and the height of the cell in the direction perpendicular to this side. In addition, we require $q = 0$ if the volume of the cell is increassing. This is sometimes necessary to avoid instabilities. The new pressure is determined by an implicit equation stating conservation of the internal energy during the time step

$$\Delta(\frac{1}{\gamma - 1} P \times V) = \Delta E_i = -(\frac{P(t) + P(t+\Delta t)}{2} + q) \times \Delta V.$$

The accelerations of the nodes are computed considering the sides of the cells as some sort of rigid walls that transfer differential cell pressures to the nodes. The mass of each node is obtained by adding some fraction of the mass of the all cells touching the same vertex. We have tested two methods to distribute the mass of the cell over its vertices: in three equal parts, or in parts proportional to the initial angle. We choose the second alternative because it provides better behaviour near the symmetry axis in some cases. Finally, we distribute the increment of the internal energy over the nodes by dividing it into three parts; equal if the increment is positive, or proportional to the temperature of the vertices (raised to the fourth) if the increment is negative. The method described above has been proven to be stable (we do not allow the time step be larger than some fraction (0.5) of the Courant limit: $\min(l_2/c_s)$), robust, and sufficiently accurate for our needs. (We have run several cases for which analytical solution exists). The code allows for subcycling. Sometimes the mentioned limit in time step is smaller than the value appropriate for solving the radiation transport. If one takes simply the minimum of both, the radiation transport, the most time consuming part of the code, would be computed more often than necessary. In such situations, we perform several hydrodynamic substeps (below some predefined maximum) between radiation timesteps. We simply add successive values of ΔE_i, and then apply the total to the energy equation.

1.5 Eulerian Hydrodynamics

Lagrangian hydrodynamics is appropriate for solving ICF problems, because it allows for large differences in scale length and density between different regions (dense thin foils versus rarefied plasma

corona). In addition, it automatically takes care of contact discontinuities at interfaces between different materials. But in some situations, in particular, when several flows collide or converge, the mesh becomes severely distorted. This has two consequencies: a loss of accuracy, and, what is even worse, some cells may become so thin, that the maximum timestep goes to zero, stopping in practice the code. We have adapted our scheme to become an arbitrary Eulerian Lagrangian (AEL) one. After a Lagrangian time step is performed, we allow for the possibility to project the obtained solution onto an arbitrary new mesh. The Eulerian method is just a particular case, in which one chooses the new grid equal to the old one. To easily implement the projection process, we rectrict ourselves to the case in which the new mesh has the same topology (one to one correspondence between cells), and is very close to the old one. In addition, both meshes have to have the same external and internal (interfaces between materials) boundaries. Magnitudes that are defined at the nodes, as the velocity, are simply linearly interpolated. The new mass in each cell is computed adding the fluxes through the three sides to the old one. The fluxes are computed as the swapped volume times the density of the donor cell. For the temperature, we have tested two methods: linear interpolation, and a method similar to the one used for the density. Both give very similar results. We use the second one in the current version of the code, because it conserves internal energy. The methods for generating the most appropriate mesh are out of the scope of this work. At the moment. we use some simple, rather intuitive criteria, for example, we inhibit node displacement in some directions, or prescribe a smooth transition between the Lagrangian and Eulerian mesh in certain regions.

1.6 Time Advance

In the previous section, we have described how to compute the deposition of energy at the nodes. In addition to radiation transfer and hydrodynamic work, energy deposition by ion or laser beams is included in the code. One might be tempted to advance the temperatures by a simple formula of the form

$$C_j \Delta T_j = Q_j \Delta t,$$

where C_j is the thermal capacity of the node (computed in a similar way as the inertial mass), and Q_j is the total deposition. In the cases of interest, in which Q_j is of diffusive origin (at least in some region), the above explicit scheme is unstable for any practical Δt. One possible solution would be an implicit scheme, in which Q_i is computed using temperatures at time $t+\Delta t$. However, these are unknown before the time step is performed. This leads to a system of coupled equations that has to be solved. In our case, temperatures are coupled indirectly through radiation intensities ($\Gamma's$) at the interfaces. These would have to be included as unknowns in the system of equations. In a typical simulation with 8000 cells and 32 beamlets, there will be about 250000 simultaneous equations, and 8 Mbytes of memory are required to store the coefficients. Things become even worse if frequency resolution is desired. These difficulties can be overcome by using the symmetrical semi-implicit (SSI) scheme (Livne). The basic idea is the following: the instabilities result from the fact that an overstimated (understimated) temperature gives place to a large negative (positive) deposition, which then produces even larger understimated (overstimated) values in the next timestep. The error increases exponentially. Taking into account the temperature dependence of Q_i, one can write

$$C_i \Delta T_i = (Q_i + \frac{\partial Q_i}{\partial T_i} \Delta T_i) \Delta t.$$

This stabilizes the scheme, but energy is not conserved. In case that the values of Q_i have been computed consistently: $\sum Q_i \Delta t = \textit{External Sources} \neq \Delta E_i = \sum \Delta C_i T_i$. In order to obtain energy conservation, we simply store the energy error in each cell ($\equiv -\partial Q_i / \partial T_i \Delta T_i$) and add it in the same cell as a energy source in the next timestep (In the original paper of Livne, the energy error is slighty smeared across the cells, but we do not find any reason to do it in our code). The equations we really use are

$$C_i (T_i^{n+1} - T_i^n) = (Q_i + \alpha \frac{\partial Q_i}{\partial T_i} (T_i^{n+1} - T_i^n)) \Delta t + \beta \epsilon_i^n$$

$$\epsilon_i^{n+1} = \epsilon_i^n + Q_i \Delta t - C_i(T_i^{n+1} - T_i^n),$$

where α and β are free parameters. We use $\alpha = 1$ and $\beta = 0.5$ in our computations. Taking $\alpha = \beta = 0$, the explicit method is recovered. The first equation guarantees stability, while the second states energy conservation. The scheme is thus stable and consistent, and is converging to the true solution for $\Delta t \rightarrow 0$. We have made several one-dimensional runs to check this method. We solved a non-linear heat wave in absence of hydrodynamic motion, and we compared the results with the ones obtained by using an implicit method and with analytical solutions. The number of time steps necessary to obtain a given accuracy is two times larger using SSI. But because no system of coupled equations has to be solved, the performances of both methods are similar. To apply SSI, the values of $\partial Q_i / \partial T_i$ are needed. Because evaluating themt exactly would be rather cumbersome, we compute aproximations in the optically thin (volume radiator) and thick (thermal conduction) limits and take the minimum of both. In addition, we consider the interface between an optically thin and an optically thick cells as a freely radiating surface. A disadvantage of the SSI method should be mentioned: it is very difficult to find a criterion to automatically determine the timestep. For example, if one prescribes a maximum variation of T, it can happen that SSI can not be satisfied for any Δt, due to the $\beta \epsilon_j^n$ in the above equations. Occasionaly, we find that, although the SSI method is stable by itself, the set of SSI equations combined with the Δt equations is not. In such a cases, we have to force ad hoc bounds to values of Δt. Finally, the existence of ϵ_i can be used in several places of the code as a common energy pool. For example, if somehow a cell gets a negative temperature, it can be forced to be zero, and the corresponding energy stays at ϵ_i.

2. Description of the Code

2.1 General Organisation

The scheme described in previous sections has been implemented in a computer code written in standard C and should run on any UNIX system. So far, it was tested on IBM RISC and CRAY YMP computers. To run the code under other operating systems would require some modifications, that we estimate small. The source code is distributed in several files. The ones with suffix *.h* are include files with declaration of types, parameters, global variables and structures, the ones with sufix *.c* are usual sources, and the ones with sufix *.data* are user input as described below. The code needs as input like the mean free path of radiation (λ) depending on density, temparature and material. In order to have a maximum of flexibility, we use C macros that the user must define in files with suffix *.data*. For example, the file *radia.data* can contain:

```
......
#define freepath(m,r,t) 0.001
......
```

In this case, $\lambda = 0.001$ independent of material number, density and temperature. A more complex definition can be used:

```
......
#define COEF 3.245
#define freepath(m,r,t) (m==1?0.1:COEF*t/r)
......
```

Now, $\lambda = 0.1$ in regions filled by material with number 1, and $\lambda = 3.245T/\rho$ in other places. It is possible also to use C functions:

```
......
real freepath(m,r,t)
real m,r,t;
{
   real l;
   if(m==1){
      l=0.1;
   }
   else{
      l=3.245*t/r;
   }
   return(l);
}
......
```

with the same results as before. Source files contain *include* statements refering to these files. Obviously, a change in one of these macros requires recompilation and generation of a new executable code. In practice, this is no problem, because the computer time required is several seconds, very small in comparison with a typical run time, of at least several minutes. In addition to files mentioned, there are files with sufix *.dat* corresponding one to one with the ones with suffix *.data*, but containing some test case. They are intended as documentation aid. The program builds up automatically using the information in file *makefile*. File *FILELIST* contains a list of other auxiliar and documentation files. All files are grouped together in an archive file that can be managed by the Unix comand *ar*.

2.2 Input/Output

The code reads initial conditions and writes results in files organized as follows. The filename

is just the time (in C format "%g") that corresponds to the information stored in it (0, $1e-09$, or $1.3e-06$ are legal filenames in Unix). The first four bytes are the ASCII values of the string "MF02" (a sort of magic number), the rest is a list of floating point numbers, each coded in four bytes in standard format (xdr). To each of these files corresponds another file with the same name suffixed by *.d* that describes the contents of the first (*0.d*, *1e − 09.d*, etc ...). The first line is the string "MF01" (another magic number). The following lines (separated by 'newline' characters) have the following format: in columns 1 to 16, the name of the variable adjusted to the left (C format "%-16s"), in columns 17 to 24, the position of the first value in the file (begining with 1) (C format "%8d"), in columns 25 to 32 the number of values (C format "%8d"). In addition to this file format, the program writes at the end (appends) of the file *outlist*, scalars like timestep, total energy, etc... each preceded by its name and a blank character. It is easy to extract information from this file, for example the unix pipeline

```
grep dt outlist | cut -d" " -f2 >abc
```

creates a file *abc* with a list of the timesteps used by the program, and can be used to generate histograms, graphics, etc ...The code reads, from standard input, as the first line the name of directory (terminated by "/"), from which initial conditions have to be read and where results are written, and in the following lines: filename/time with initial conditions and filename/time where to write results. For example:

```
/home/mpqibmr/rfr/m/c/2.3.7/
0 1e-12
1e-12 3e-12
```

runs the code from time 0 to *3ps*, results are stored at 1 and *3ps*. With

```
/home/mpqibmr/rfr/m/c/2.3.7/
1e-12 2e-12
```

a restart is made from *1ps*, and results are stored at *2ps*. The standard output is used mainly for diagnostic messages.

2.3 Subroutines

2.3.1 multi.h

This file contains a list of definitions used at other places of the program. We use the CGS system for all magnitudes, except for the temperature that is given in eV. We define the type *real* as *double* (if single precision is desired, the only change in the code is to redefine it to *float*).The derived type *point* is composed by a pair of *reals*, and used to store vector quantities. The structure type *trngle* defines a cell, it contains the numbers (indexes) of the three vertices, and the three sides. The structure type *wall* defines a side between two triangular cells, it contains the indexes of the two common vertices, and the two cells. In the case that a side belongs only to one cell (at the boundary), the other index takes value -1. Cells, nodes, and sides are numbered starting from 0. The parameters NTMAX, NPMAX, and NWMAX, are the maximum allowable values, respectively. We define two principal structures: *mesh* and *hydro*.

mesh is intended to store fixed data that do not change during the computations. It contains the actual total number of cell, nodes, and sides, as well as pointers to arrays of initial coordinates (type *point*), triangles (type *trngl*), sides (type *wall*), and initial angles.

hydro is intended to store the actual values of variables. It contains densities, surfaces, volume, material code, and gas constant at the cells, and coordinates, velocities, temperatures, energy errors, inertial masses, and thermal capacity at the nodes. Also, it contains the timestep for the next interval.

We give an example for how to use of these structures. Let us assume that m and h are pointers to structures of type *mesh* and *hydro*, respectively,

`h` is a pointer to *hydro*

`h->T` is a pointer to an array of temperatures
`m` is a pointer to *mesh*
`m->t` is a pointer to *trngl*
`m->t[3]` is a *trngl*
`(m->t[3])->p` is a pointer to an array of indexes
`(m->t[3])->p[0]` is a the index of the first corner of the fourth cell
`(h->T)[(m->t[3]) ->p[0]]` is the temperature at the first corner of the fourth cell.

Finally, it contains definitions for more specific structures, global variables, and some macros (*min* and *max*).

2.3.2.main.c

Basically, this file contains flow control routines. *main*() opens files, reads fixed information, creates structures, and enters the most external loop: in each trip, a set of initial conditions are read, integration is done, and results are written. The integration process takes place in an internal loop: a timestep is performed, if successfull, the time interval is increased or decreased, depending on the value of the variable *control*, if it fails, the results are rejected, the time interval is decreased, and a new try is made. This process continues until the final time is reached. Actually, the timestep is performed by the routine *timestep*. First, the hydrodynamical variables are advanced by *fluid*(), the radiation deposition and its temperature derivative are computed by *radia*() and *derdepo*(), respectively, the deposition of energy is computed by *ions*(), the energy deposition in the cells are distributed over the nodes, the equation of energy is solved by *ssi*(), and the results are changed to an arbitrary Eulerian Lagrangian (AEL) mesh by *kami*(). The flags HYDRO, RADIA, IONS and EULER, allow for switching on or off these processes, individually. In the actual version, the timestep can fail only due to hydrodynamic reasons (too many subcycles, or excessive density change). The temperature variation is computed by *check*(), and compared against a nominal value. The relation of both is returned by variable *control*, that will reduce the next timestep in case of excessive variation. Also the relative number of subcycles is taken into account to compute this variable.

2.3.3 radia.c

The first three routines: *activeradia*(), *getbtotal*(), and *getbdown*(), of this file are used to initiate the structures defining the boundary: *btotal*, and *bdown*. The first includes the whole boundary: all the sides that belong to one cell. The second is a subset of the first: the sides with end points r equal to the minimum value of r. In this part, reflecting boundary conditions will be applied. The routine *radia*() implements the major part of the numeric treatment of radiation as described before; it is in some sense, the core of the program. It has three nested loops. The two external ones go through the differents beamlets, the internal one through the cells. The deposition of energy at the nodes is returned in *depoP*, and the deposition at the cells in *depoT*. The total deposition is the sum of both. The radiation incident on the boundary is computed by *radcond*(), and the temperature derivative of the deposition at the nodes by *derdepo*(). The routine *beamgeo*() computes several geometric quantities of the cells (height, depth, ...). To determine the radiation intensity for one beamlet is necessary to sweep the cells in some order, depending on the direction of propagation. *beamsort*() generates the sorted list of cells.

2.3.4 fluid.c

The routine *fluid*() controls the hydrodynamic timestep. It starts calling *fluidinitTC*() to get temperatures at the cell centers from temperatures at the nodes. Then it enters into the subcycling

loop. Routine *fluidsub*() computes the maximum allowable timestep, and advances density, coordinate, velocity, and cell temperature as previously described. After each cycle *fluidcheck*() computes the maximum variation of density. If a specified threshold is exceded, *fluid*() returns a zero, indicating failure. This process continues until a maximum allovable number of subcycles is reached (failure) or the time interval is completed. The hydrodynamical energy deposition at the cells is computed from the initial and final cell temperature, and is returned in *depo*. The variable *hydrocontrol* returns a value that will be used by *timestep*() (in *main.c*) to determine the timestep used in the next interval.

2.3.5 ions.c

Routine *ions*() determines the power deposited in each cell by an ion-beam. This is modelled as a discrete number of straight rays. For each of them, an entry point is first located, and from there, rays are followed through the mesh, until they either exit the mesh or are attenuated completely. No momentum deposition is taken into account. This routine can also be used for laser deposition, provided that refraction effects are not considered to be important.

2.3.6 kami.c

The routine *kami*() implements the projection of the values of the variables defined in a mesh $h2$, into a slightly different one $h1$. It proceeds as described previously.

2.3.7 mesh.c

This file contains routines related to *mesh* structures. *getmesh*() generates such a structure. It calls first *readmesh*(), to read the initial values of the coordinates, and an array of size 3×number of triangles, that contains for each triangular cell the number (starting from one) of the nodes corresponding to its vertices. Once this information is ready, *ccwmesh*() is called to make sure that no cell has zero volume, and to order the vertices of each cell in counter-clock-wise sense. Then, *weighmesh*() computes the initial angles used later to distribute cell mass and thermal capacity. Finally, *endmesh*() is called to generate the data structures with information about the interfaces between cells.

2.3.8 inout.c

This file contains subroutines to manage *hydro* structures, which contain the fluid variables. *hydronew*() generates an empty structure of this type. *hydrocopy*() copies the contents. *readvar*() reads data from a binary file into some members. *initmass*(), *initvols*(), and *initthermo*() complete the contents by computing the inertial mass at the nodes, the area and volume of the cells, and the thermal capacities of the nodes, respectively. *writemesh*() writes a binary file from the data in one structure.

2.3.9 store.c and mf.c

The routines in this files: *store*(), *storein*(), *storeout*(), *mfputreal*(), *mfopen*(), *mfput*(), *mfget*(), and *mfclose*(), implement the input/output to files in the format described previously. They are called by the routines in *mesh.c* and *inout.c*.

2.3.10 main.data

This file must contain the following definitions used by the main program:

TVARMAXLIM	Temperatures are truncated if their absolute variation in a timestep is larger than this VALUE
TVARMAXREL	Temperatures are truncated if the relation $T(t+\Delta t)/T(t)$ is smaller than this value
DTmax(t)	Maximum allowable timestep as a function of time
DTmin(t)	Minimum allowable timestep as a function of time
PRINTFLOW	Enables some values to be printed
PRINTST1	Enables some values to be printed
PRINTST2	Enables some values to be printed
RADIO	Enables radiation transport
HYDRO	Enables hydrodynamic motion
IONS	Enables ion-beam deposition
EULER	Enables non-lagrangian hydrodynamics
TEMPERATUREVAR	Acceptable variation of temperature (must be smaller than TVARMAXLIM)

2.3.11 radia.data

This file must contain the following definitions used by the radiation routines:

NTETAS	Number of intervals in θ
NPHIS(ITETA)	Number of intervals in ϕ (can depend on i)
I(eta,r,z,time)	Incident intensity at boundary as a function of θ,ρ,T and t
freepath(m,r,t)	Mean free path for radiation as a function of material, ρ, and T

2.3.12 fluid.data

This file must contain the following definitions used by the hydrodynamic routines:

XMIN	Defines the allowable limits of space coordinates
XMAX	...
YMIN	...
YMAX	...
PEXTERM	External pressure
GX	Gravitational acceleration
GY	...
PRINTHYDRO	Enables some values to be printed
MAXSUBCYCLI	Acceptable number of subcycles
MAXRELATIVE	Maximum relative variation of density
RELATIVEVAR	Acceptable relative variation of density
RECTANGLES	If set, the code termalizes even against uneven cells before hydrodynamic timestep (used to enforce symmetry of the mesh in some cases)
SCHMUTZIGET	Code to be inserted inside *fluidsub*() (used to include additional constrains to the motion)

2.3.13 ions.data

This file must contain the following definitions used by the ion-beam deposition routines:

DeDx(e,m,r,t)	Stopping power ($\frac{\partial e}{\partial l}$) as a function of e, material, ρ, and T
NBEAMS	Number of rays to be used
beampower(i)	Power in ray number i
P0X(i)	Initial position of ray number i
P0Y(i)	...
N0X(i)	Vector in the direction of propagation of ray number i
N0Y(i)	...

2.3.14 EOS.data

This file must contain the following definitions used to define material properties

A(mid)	Atomic mass number as a function of material
Z(mid)	Ion number as a function of material

2.3.15 ssi.data

This file defines parameters for the SSI method

ALPHA)	1.0
BETA)	0.5
GAMMA)	0.0

2.3.16 Euler.data

This file must contain a C routine to be used to compute the new mesh after one timestep has been performed. The calling format is

$$void\ newmesh(m, h1, h2, h3);$$
$$mesh\ *m;$$
$$hydro\ *h1, *h2, *h3;$$

where $h1$ is the former mesh, $h2$ is the mesh after the lagrangian timestep, and $h3$ is the mesh to be generated

2.4 Initial Conditions

To run the code the binary file corresponding to time zero must contain at least the following variables:

NAME	NUMBER	DESCRIPTION
np	1	number of points
nt	1	number of triangles
x	np	initial x coordinates
y	np	initial y coordinates

pt	3×nt	topology of the mesh

On the other hand, the binary file corresponding to the time at which integration begins (can be the same as above) must contain

NAME	NUMBER	DESCRIPTION
x	np	x coordinates
y	np	y coordinates
vx	np	x components of the velocity
vy	np	y components of the velocity
T	np	temperatures
GE	np	errors in the energy
rho	nt	densities
mid	nt	material codes
dt	1p	time interval to be tried

2.5 Graphic Pre- and Post-Processors

Even if the computer code, as described in this report, is consistent from the numerical point of view, it is not trivial to run it, due to the large amount of input and output data. (several Mbytes in a normal case). We have developed an additional software package, in order to provide easy access to the capabilities of the code. Three categories of tasks have to be carried out:

i) Generation of (binary) data files containing the initial values of the variables.

ii) Interactive graphic visualization of the results, and production of hard copy plots.

iii) General management of the system: automatic compilation, creation and destruction of cases, starting and stopping runs, etc...

We use an interpreted computer language (r91) that runs under Unix, has access to X-Windows functions, and generates Postscript color output. We developed it for an IBM RISC/6000 system with a high resolution color display ($1000 \times 1250 pixels \times 8 planes$). Use of different hardware would require, in principle, only minor changes.

2.6 File Distribution

The whole system must be installed in some directory. Three subdirectories are needed:

r91 contains the interpreter program (executable module called *r91*), the interpreted programs (in files with prefix *r91.*), the graphic output (in files with prefix *km.*, intended to be of internal use only, or *kp.* in the case of Postscript files). The file *FILELIST* contains a brief description of the main files.

v contains the different versions of the code, each installed in its own subdirectory. **v/2.3** correspond to the one described in this report. It contains basically source files, and eventually compiled ones. Other versions can be created for specific purposes or for further development of the code, and coexist with older ones. Old runs can be in any moment reproduced without been affected by bugs or modifications introduced later.

c contains cases, each in a subdirectory with his name. Initially a case contains the definition of a specific problem in the following way: files with suffix *.data* contains definitions of macros and

procedures, except *version.data*, that contains the name of the version of the code to be used. *r91.gm* is a r91 program able to generate the binary files with the initial values of the variables and the mesh topology. Finally *rtimes* is a list (one value in each line) of instants of time for wich results are to be stored. Later, other files are generated: *a.out*, the executable module, 0 and *0.d*, the initial conditions, and *control*, a file to be read as standard output. Once the code has been run, it constains results files (with names like $1e-09$, $1e-09.d$, $2e-09$, $2e-09.d$, ...), *output* with diagnostic information (including required CPU time), and *outlist*, mentioned before.

2.7 Computer language r91

This language is an interpreted one, that is, an executable module (whose source code is written mostly in C) must be loaded prior to execute r91 programs. These are usually stored in text files, and loaded, translated to intermediate representation, and executed automatically. As any other interpreter, it is not very efficient in terms of computer resources, but allows fast and easy development of interactive graphic software, that otherwise would be prohibitive in terms of human resources. It uses vectorial variables consisting in zero, one or several numbers or characters, typical assignations would be:

$a =$ "*hello you*\n";
$b = 1 : 4 : 8 : 12;$

It understands escape codes in the usual format in C. The statements must be ended by ";". Expressions can contain a mixture of scalars (variables of size one) and vectors. Two vectors are operated element by element, but if one of them is a scalar, its value is used in all the operations. Except in this case, the same length of operands is usually required. Operation between vectors and strings are allowed, if necessary the string is previously converted into a vector. If $a = 5$, $b = 3 : 5 : 2$, and $c = 1 : 3 : 1$

$x = b + c$; (assigns to x the value $4 : 8 : 3$)
$y = a * c$; (assigns to y the values $5 : 15 : 5$)
$z =$ "AB" $+ 1$; (assigns to z the values $66 : 67$)

In addition to arithmetic oparators (+,−,*,and /), comparison operators (==,>,<,<=,>=,and !=), and logical ones (!(negation), &(and) and |(or)), the following operators are available:

Concatenation (both variables can have different lenght).
$x = a : b$; (assigns to x the values $5 : 3 : 5 : 2$)

Ellipsis (both variables have to be scalars)
$y = 4...2$; (assigns to y the values $4 : 3 : 2$)

Cardinal. Returns a scalar with the number of elements in a vector.
$z = \#b$; (assigns to z the value 3)

Selection. Is the extension of array index to vectors
$x = b[c]$; (assigns to x the values $3 : 2 : 3$)

Binary assignment. It is treated as any other operator
$x = y = z = a$; (assigns the value 5 to x, y and z)

Ternary assignment. It requires three operands.
$m[3 : 4] = 5 : 2$; (assigns to the third and fourth elements of m, the values $5 : 2$)

Address. It is intended to be used only to interface directly C routines. It returns the physical

address of a permanent variable, coded in four bytes of a string.

$z = \&a$; (can result in z having the value "9&%b")

The syntax to calling a function is the usual one in computer languages:

$x = power(c, 2)$; (assign to x the values $1:9:1$)

Each operator has a precedence that determines in which order operations are performed ($3*2+1$ evaluates to 7, not to 9). The use of parenthesis overrides this order ($3*(2+1)$ evaluates to 9). The language allows for structured programing with *if*, *while*, and *for* constructs

```
if(a > 3){
        b = a + 1;
}
else{
        b = a - 1;
}
```

```
while(a < 3){
        if(b[a] == 0)break;
        a = a + beta(a);
}
```

```
for(i = 0; i < 10; i = i + 1){
        j = i * i + j;
}
```

Other structures are not standard. The *trap* construct allows to bound the effects of an error. If an error occurs (f.e. division by zero, or a call to an non-existent function) inside the braces, the execution is reassumed at the first line outside. For example

```
trap{
        flag = 1;
        nlm(alpha, beta);
        flag = 0;
}
if(flag)print("function nlm does not exist");
```

The *run* construct permits to execute a string as a fragment of code

```
a = "c = 23" : "12;";
trap a;
```

Assigns the value 2312 to c. The function or routines return always some value (by default a zero), and can be of three types:

a) C library routines accessed directly. The arguments must be coded to the internal representation of the computer (by means of & operator, or funtions *toI*, *toD* and *toF*), a typical example is

```
system(&"cat filea > fileb");
```

b) Internal routines devellopedspecially for this code. A list is given in the next section
c) Routines defined in *r91* as in the following example

```
define minimum(a, b)
{
        r = a * (a < b) + b * (a >= b);
        return(r);
}
```

If *return* is not present or has no argument, a zero value is returned. The variables beginning with lowercase are local to the function. The ones begining with uppercase are global to all funtions, and its value is preserved. A function can be called by itself, for example

```
define factorial(a){if(a == 1)return(1); else return(a * factorial(a - 1)); }
```

When a funcion is found for the first time, it is searched in a table of defined functions (of any type). If not present, the file with the name of the function prefixed by *r91.* is read, and loaded. If the function is still not defined, an error results. Thess files can contain several functions separated by a white line. The above process can have the side effect of loading other functions present in the file. This is useful for creating libraries: once a call to a dummy function is done, the whole set of routines are available. To start the interpreter, a first routine (without arguments) must be called. The name is given in the input line. By default the routine *start* is called. It contains a small program that executes line to line the user input.

```
define start(){
        while(1){
                print(" > ");
                trap run input();
        }
}
```

2.7.1 r91 Internal Routines

The following routines coded in file BLIB are of general purpose

print(a)	prints a vector or a string
input()	read a string from the standard input
char(a)	converts a vector in a string
numero(a)	converts a string in a vector
sel(a,b)	returns the elements of *a* for whitch *b* is true
floor(a)	mathematical function
ceil(a)	mathematical function
fabs(a)	mathematical function
exp(a)	mathematical function
log(a)	mathematical function
log10(a)	mathematical function
sqrt(a)	mathematical function
sin(a)	mathematical function
cos(a)	mathematical function
tan(a)	mathematical function
asin(a)	mathematical function

acos(a)	mathematical function
atan(a)	mathematical function
sum(a)	adds all the elements of a vector
prod(a)	multiply all the elements of a vector
min(a)	minimum element in a vector
max(a)	maximum element in a vector
toI(a)	codes a number as an integer (in four characters)
fromI(a)	decodes an integer
toD(a)	codes a number as a double (in eight characters)
fromD(a)	decodes a double
toF(a)	codes a number as a float (in four characters)
fromF(a)	decodes a float

The following functions contained in the file GLIB implement the interface with the XWindows. Graphical coordinates and dimensions are normalized with the width of the window.

xstart(a)	opens a window of witdh a_1 and height a_2
xstop()	closes the window
xcreatemap()	creates a colormap (to be used in 4 bits screens only)
xmap(a)	defines colors
	a_1,a_2,a_3 are the red, green, and blue component of the first color
	a_4,a_5,a_6 are the red, green, and blue component of the second color
	...
xclip(a)	cliping is done outside the rectangle of corners (a_1,a_3) and (a_2,a_4)
xfunction(a)	if $a = 0$ graphic primitives are ORed against the background
	Useful to implement a zoom
xflush()	The picture is refreshed
xsettex(a)	set properties of text
	a_1 is the size of characters
	a_2 is unused
	a_3 is "L","C" or "R", for left, center or right alignment
	a_4 is "D","C" or "U", for down, center or upper alignment
	a_5 is the color
xsetline(a)	set properties of lines
	a_1 is type (0 if continuous)
	a_2 is the color
	a_3 is the thickness in pixels (0 is faster)
xsetarea(a)	set properties of fill regions
	a_1 is the color
xline(a)	plots one or more lines
	a_1,a_2 are the coordinates of the first point, $a_3 = 0$ if a new line begins
	a_4,a_5 are the coordinates of the second point, $a_6 = 0$ if a new line begins
xarea(a)	plots one or more poligons
	a is coded as for lines
xtext(a,b)	draws string b in point of coordinates a_1,a_2
xsize()	returns width and height of the window
xsend(a)	sends a message (string) to the server
xreceive()	reads a message from the server
xgin()	graphic input. returns three values: two coordinates and:
	-1 if the mouse key 1 has been pressed
	-4 if the mouse key 2 has been pressed
	-2 if the mouse has been moved while a key is pressed
	-3 if a mouse key has been released

a code between 0 and 256 if an ordinary key in the keyboard has been pressed
a code between 1001 and 1005 if an arrow key has been pressed
a code between 2001 and 2001 if an insert or delete key has been pressed
a code between 3001 and 3012 if a function key has been pressed
-100 if the constents of the window have been lost
-200 if a new message is available

The following funtions contained in file PLIB implement the Postscript output. Graphical coordinates and dimensions are normalized with the width (bigest side) of the paper.

pstart(a)	opens a Postscript output file with name a
pstop()	closes the output file
pclip(a)	cliping is done outside the rectangle of corners (a_1,a_3) and (a_2,a_4)
psettex(a)	set properties of text
	a_1 is the size of characters
	a_2 is the rotation angle in degrees
	a_3 is "L","C" or "R", for left, center or right alignment
	a_4 is "D","C" or "U", for down, center or upper alignment
	a_5 is the color
psetline(a)	set properties of lines
	a_1 is type (0 if continuous)
	a_2 is the color
	a_3 is the thickness in pixels (0 is faster)
psetarea(a)	set properties of fill regions
	a_1 is the color
pline(a)	plots one or more lines
	a_1,a_2 are the coordinates of the first point, $a_3 = 0$ if a new line begins
	a_4,a_5 are the coordinates of the second point, $a_6 = 0$ if a new line begins
parea(a)	plots one or more poligons
	a is coded as for lines
ptext(a,b)	draws string b in point of coordinates a_1,a_2

2.7.2 r91.control

This is assumed to be the entry point for the interactive use of the code. To start this program go to the directory *r91* and type:

r91 control&

A window appears on the screen with several "keys" on it. The first one "CASE:" allows to specify a case to work with. The labels ended by "$->$" activate submenus. To go back press key 2 of the mouse or escape key on the keyboard. To exit the program press escape key several times. This program perform the following tasks

i) Create a new case.

ii) Copy definition files from other cases into it.

iii) Edit definition and output files.

iv) Select a text editor (in our configuration *vi*, *textedit*, a *xedit* are available.)

v) Print definition files

vi) Run the preprocessor to generate data files with the initial conditions

vii) Compile the code

viii) Run the code. It tries that the values of time in file *rtimes* correspond exactly with the values of time for which results are available. The executable module is run (in the background), and files deleted as necessary.

ix) Erase the results and the executable code (to try a new run with different parameters, or simply to free disk space).

x) Destroy a case.

xi) Start another programs (described in the following sections)

2.7.3 r91.ctab, r91.ttab, r91.vtab

ctab presents in a window the list of cases availables. When one of them is pressed, a message is sent to the other running programs, with the meaning "put attention to this case". When *ttab* receives such message, it displays a list of files with results (times) available for the selected case. When one of these is selected, a new message ("put attention to this time") is sent. Finally, when *vtab* has received both messages, it displays a list of variables and modificators (logarithmic scale, lagrangian/eulerian representation, ...) available for this case and for this time. When one is selected, a new message is sent. This proccedure allows a maximum of flexibility; the described windows can be placed anywhere on the screen, and can have any size. The flow of messages control the display windows as described below.

2.7.4 r91.nav2

The basic graphic presentation of 2D results is through colour plots. The tone or intensity in a given point represents the value of some scalar variable there. In addition the computational grid can be plotted, and vectorial variables draw as fields of arrows. The program *nav2* generates the window that controls the format of the plots. Colour scales, viewport, size of text, axes, background, margins, and type of line, are controled through menus. Each time a change is produced, a message is sent to the display programs. Additionally, some programs can be started also from this program.

2.7.5 r91.sup

This program creates a window where the actual drawing takes place. Several cases started from *nav2*, can run concurrently, displaying different views, variables, cases, ... Initially, a black background is presented in the plot area. Once the program has received messages specifying case,time and variable to be plotted, the plot is produced, triggered either automatically (see below), or by touching the plot area. Additional messages update correspondingly the drawing. These windows are equiped with a control menu with the following "keys":

Schloss: pressing here, the window enters into a closed state. No messages are received. Pressing it again, the normal mode is entered. This allows to select which window will receive messages, and thus control independently what is presented in each window. Once the normal mode is entered, a new message (f.e. selected time) affects all the windows, but produces different plots (f.e. different variables at that time) in each.

Auto enables or disables the automatic triggering of the plot. The user must touch the plot area to produce the plot. This proves convenient when several successive changes have to been done, and the time need to redraw the plot is not negligible. Also when the plot is lost due to window resize or exposure, the user must touch the plot area to redraw the window.

Skala changes the viewport to the values specified by *nav2*.

Zoom+ changes the viewport by means of a Press-Drag-Release process.

Zoom− similar to the former, but the size of the figure is reduced.

Beweg. the plot is just translated. This, and the above three selections are implemented indirectly, through messages. All the non-closed windows are affected (including the original one).

Kopie produces a graphic metafile. The name is generated as a number (correlative) prefixed by *km*.

2.7.6 r91.doc

This programs acts as an intermediate processor for graphic output. It loads graphic metafiles (ussually produced by *sup*, for example), some text, lines, ...can be introduced, and output generated in metafile or Postscript format. The elementary use is to press LOAD, type the input file name (without prefix), press POST, and type the output file name (without prefix), repeat the former cycle as needed, and press escape key to exit. Once postscript files are produced use *lpr* Unix comand to send them to some printer.

2.7.7 r91.gtab, r91.rtab

These programs present lists of graphic metafiles, and processes running in the background, respectively.

2.8 Install procedure

Let us assume that files R91V19 and M92V23 in the current directory are archive files containing the graphic processor and the 2D code, respectively. Type

```
mkdir r91 c v
cd r91
ar x ../R91V19
make
cd ..
```

to install the system (it take about five minutes),

```
cd v
mkdir 2.3
cd 2.3
ar x ../../M92V23
cd ../..
```

to install the 2D code,

```
cd v/2.3
FALL
2
2.3.1
cd ../..
```

to install an example (named 2.3.1) case, and

```
cd r91
```

r91 control&

to start operations. Good luck !!!

Appendix: Opacity Parameters

The table below gives typical values of the mean free path for radiation, obtained by fitting power laws to SNOP computations (Tsakiris). The mean free path is related to the opacity κ by $\lambda = 1/\kappa\rho$, and this approximated by $\kappa \simeq aT^s\rho^r$, (κ in cm^2/g, T in ev und ρ in gr/cm^3). This expression is valid for temperatures between $30eV$ and $1KeV$ and densities between $0.1g/cm^3$ and $10g/cm^3$. The coeficients are

(Rosseland opacity)

Material	a	s	r
Al	3.78	-2.482	0.481
Ti	7.19	-2.209	0.386
Fe	9.74	-2.268	0.314
Cu	13.89	-2.207	0.295
Mo	67.42	-1.493	0.222
Sn	72.19	-1.571	0.160
Ba	81.34	-1.619	0.142
Eu	129.30	-1.450	0.094
W	244.12	-1.119	0.005
Au	279.90	-1.058	0.001
Pb	290.54	-1.047	0.000
U	295.05	-1.145	0.037

(Planck opacity)

Material	a	s	r
Al	34.18	-2.415	0.483
Ti	85.84	-2.071	0.436
Fe	89.97	-2.131	0.377
Cu	100.81	-2.120	0.355
Mo	315.86	-1.563	0.308
Sn	328.55	-1.588	0.228
Ba	324.04	-1.644	0.244
Eu	413.90	-1.536	0.238
W	646.40	-1.225	0.199
Au	666.04	-1.233	0.165
Pb	645.51	-1.270	0.155
U	590.61	-1.415	0.194

The following table has been obtained by fitting power laws to data available from the SESAME library (Murakami). λ is aproximated by an expresion in the form $aT^b\rho^c$ with λ in cm, T in ev und ρ in gr/cm^3. The coefficients are:

Material	a	b	c
CH	$1.0 \cdot 10^{-12}$	4	-2
Al (Rosseland)	$8.7 \cdot 10^{-9}$	2.5	-1.5
Al (Planck)	$1.8 \cdot 10^{-9}$	2.4	-1.5
Au (Rosseland)	$6.0 \cdot 10^{-6}$	1.0	-1.0
Au (Planck)	$3.0 \cdot 10^{-7}$	1.2	-1.2

Bibliography

Zel'dovich and Raizer, Physics of Shock Waves and High-Temperature Hydrodynamic Phenomena, Vol. 1, Academic Press, New York, 1966.

M. Murakami, J. Meyer-ter-Vehn, and R. Ramis, Thermal X-Ray Emission from Ion-Beam-Heated Matter. Journal of X-Ray Science and Technology 2, 127-148 (1990)

G. D. Tsakiris and K. Eidmann, An Approximate Method for Calculating Planck and Rosseland Mean Opacities in Hot, Dense Plasmas. J. Quant. Spectrosc. Radiat. Transfer Vol. 38, No 5, 353-368 (1987)

R. D. Richtmyer and K. W. Morton, Difference Methods for Initial-Value Problems, New York: Wiley Interscience 1967, 2nd ed.

C. Stöckl and G. D. Tsakiris, Experiments with laser-irradiated cylindrical targets. Laser and Particle Beams, vol. 9, no.3, 725-747 (1991)

R. Ramis, R. Schmalz, and J. Meyer-ter-Vehn, Multi - A Computer Code for One-Dimensional Multigroup Radiation Hydrodynamics, Computer Physics Communications 49, 475-505 (1988)

E. Livne and A. Glasner A Finite Difference Scheme for the Heat Conduction Equation, Journal of Computational Physics 58, 59-66 (1985)

Demonstration Runs

We created a collection of typical cases with the double purpose to check the code (when some modification has been done, we run the complete set, in order to be sure that no side-effects are produced), and to allow the user to start with a configuration more or less similar to his needs. The cases are briefly described below. Prefix 2.3 refers to the actual version fo the code. The time necessary to run the code on our IBM RISC/6000 Mod. 320 is given.

2.3.1) A 100 eV thermal radiation is incident over the central region of an optically thick disk. A thermal wave propagates. Only radiation is enabled. (Fig 3., 206 sec.)

2.3.2) A two-dimensional shock tube, but with one-dimensional solution. Only lagrangian hydrodynamics enabled. (Fig 4., 75 sec.)

2.3.3) Simulation of a non-linear Rayleigh-Taylor instability. Only eulerian hydrodynamics enabled. (Fig. 5 and 6, 2959 sec.)

2.3.4) Spherically symmetric implosion of a composed pellet, driven by an external pressure. Only mixed eulerian-lagrangian hydrodynamics enabled. (Fig. 7, 160 sec.)

2.3.5) Disk irradiated by 100 eV thermal radiation. Ablation takes place, and a shock wave appears inside the target. Radiation and lagrangian hydrodynamics (Fig. 8 and 9, 615 sec.)

2.3.6) Spherically symmetric implosion of a composed pellet, driven by radiation. Radiation and AEL hydrodynamics. (Fig. 10, 1510 sec.)

2.3.7) Capilar tube irradiated from one side with 150 eV planckian radiation. Plasma collision on the axis. Radiation and AEL hydrodynamics. (Fig. 11, 12, 13, and 14, 11327 sec.)

2.3.8) Cylinder shot by a 700 Tw ion-beam. Conversion of the beam enrgy to thermal radiation. Radiation, lagrangian hydrodynamics, and beam deposition. (Fig. 15 and 16, 567 sec.)

2.3.9) Hohlraum simulation. The inside of a spherical cavity is irradiated by a laser-beam that enters through a hole. All routines enabled. (Fig. 17, 18, 19, and 20, 2412 sec.)

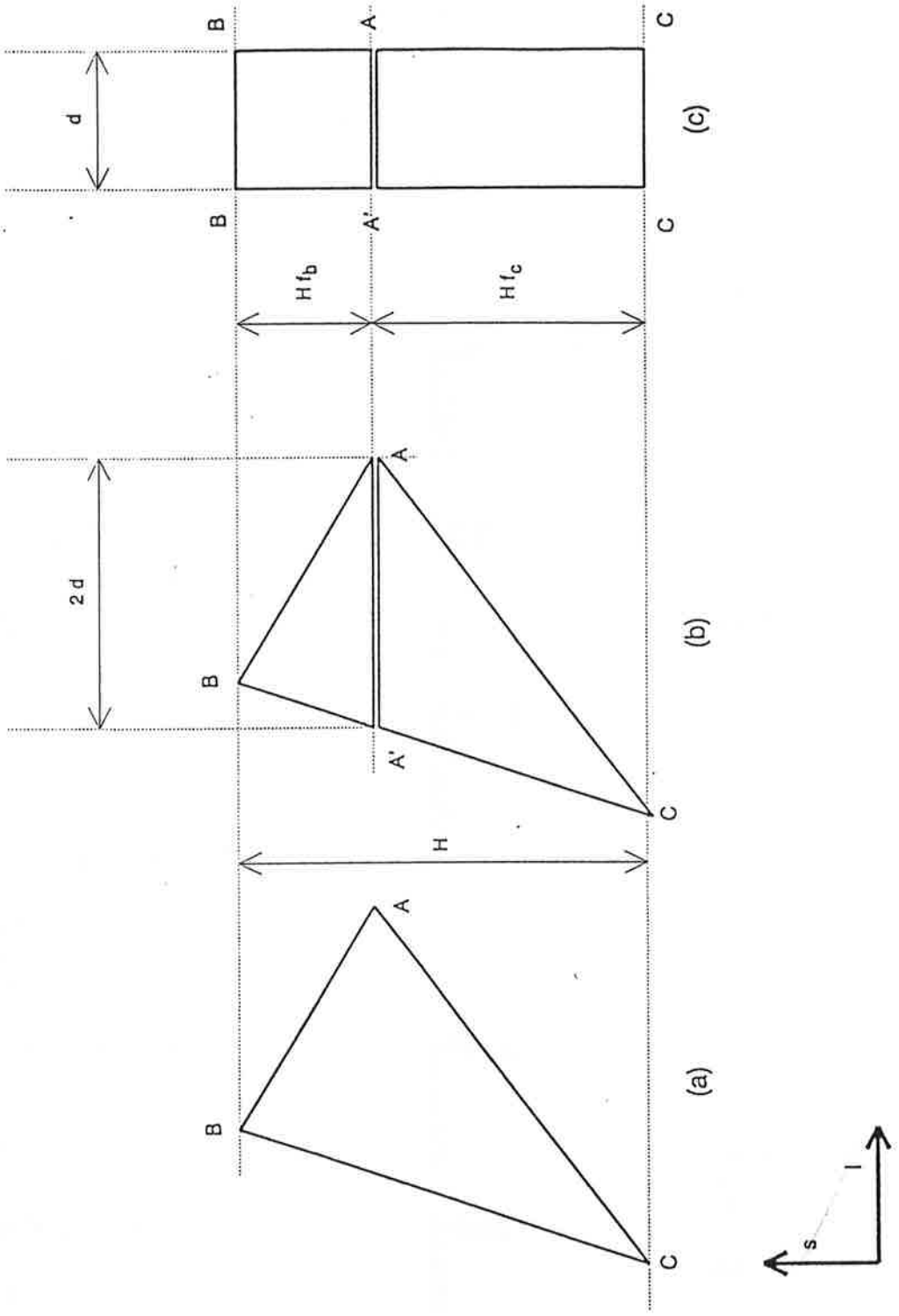


Fig. 1

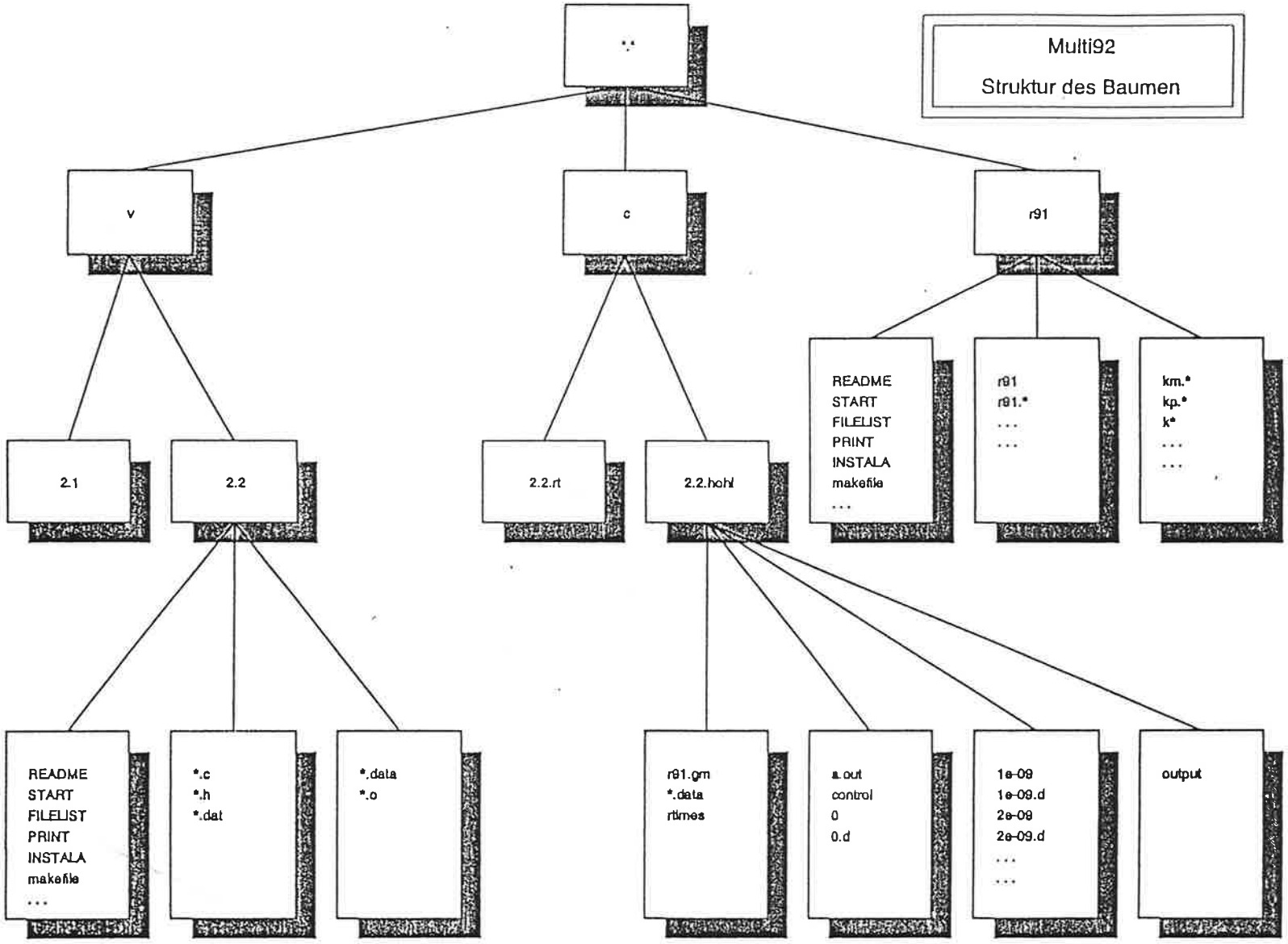
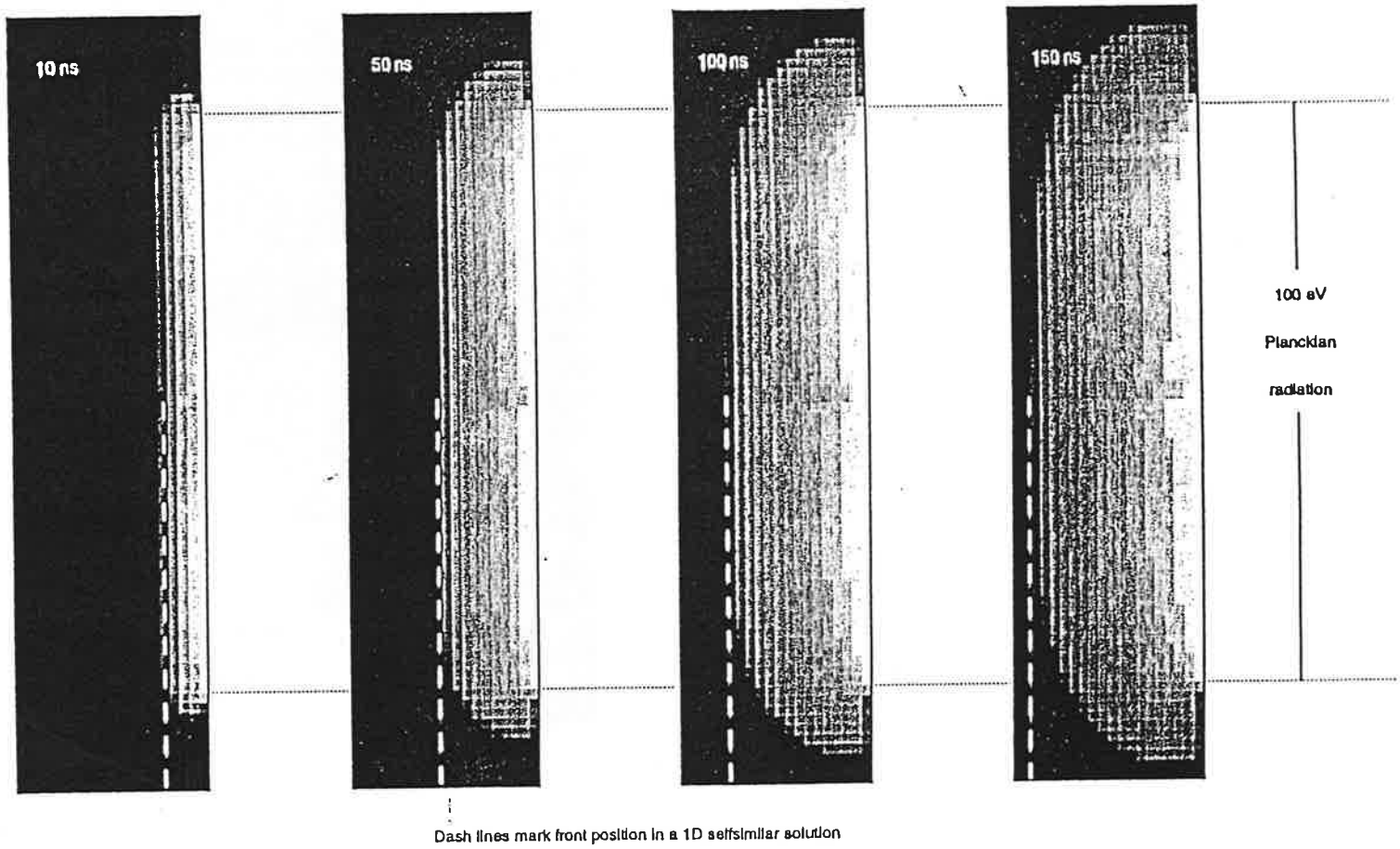

Multi92
Struktur des Baumen
v
c
r91
2.1
2.2
2.2.rt
2.2.hohl
README
START
FILELIST
PRINT
INSTALA
makefile
...
r91
r91.*
...
...
km.*
kp.*
k*
...
...
README
START
FILELIST
PRINT
INSTALA
makefile
...
*.c
*.h
*.dat
*.data
*.o
r91.gm
*.data
rtimes
a.out
control
0
0.d
1e-09
1e-09.d
2e-09
2e-09.d
...
...
output

Fig. 2

Fig. 3

Propagation of a 2D thermal wave (optical depth: 167)



Test case: 1D shock tube

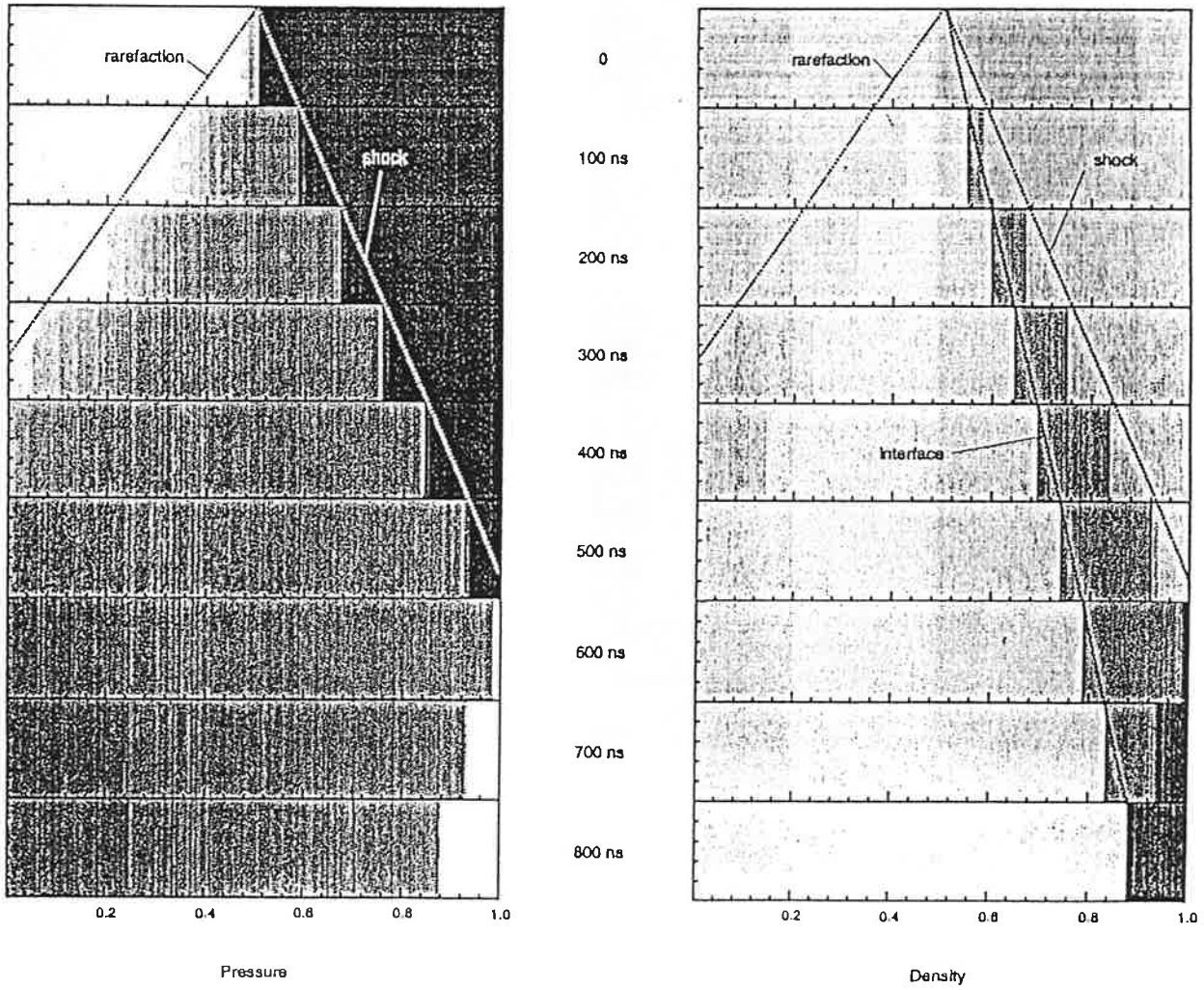


Fig. 4

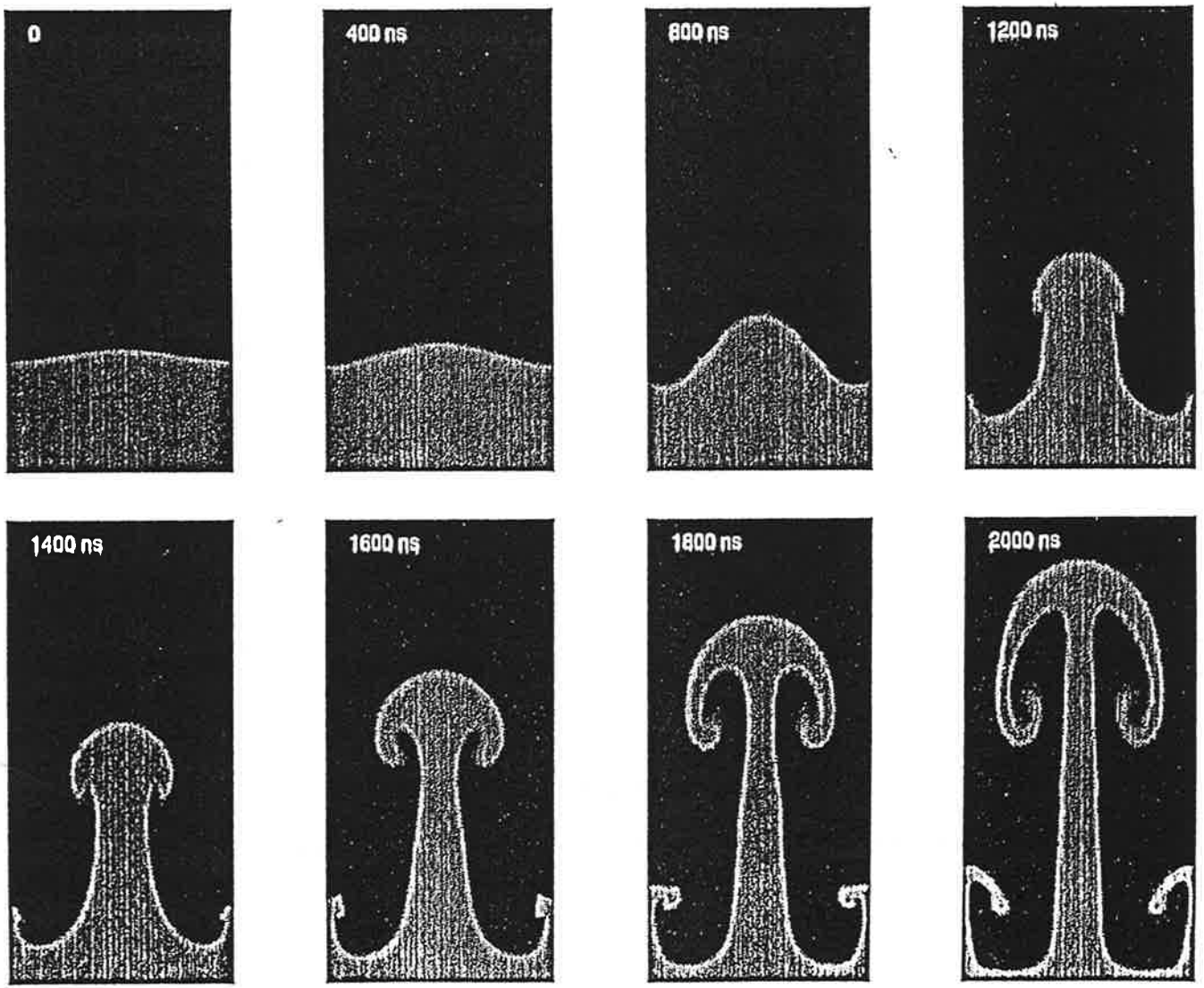


Fig. 51

Fig. 9

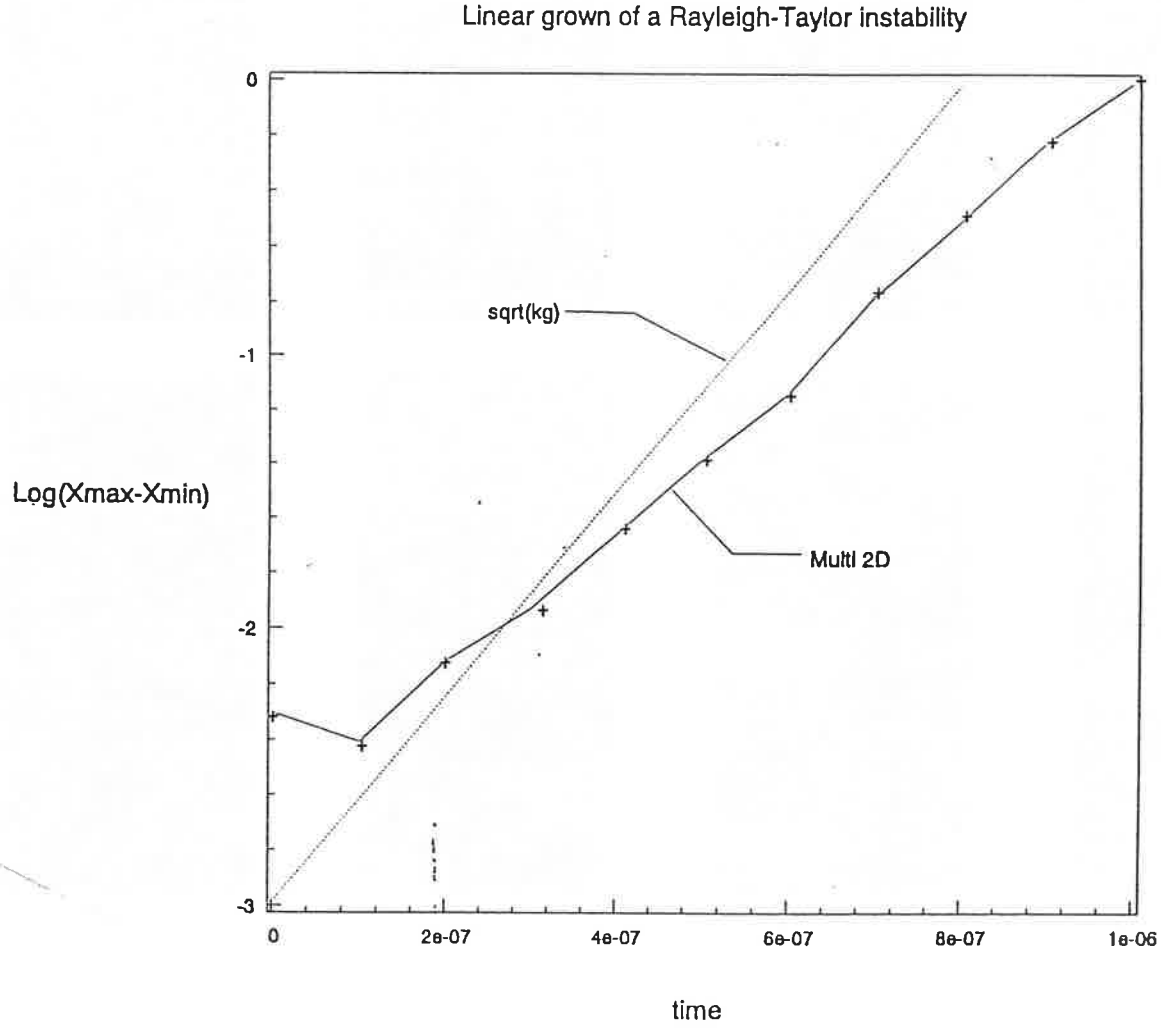
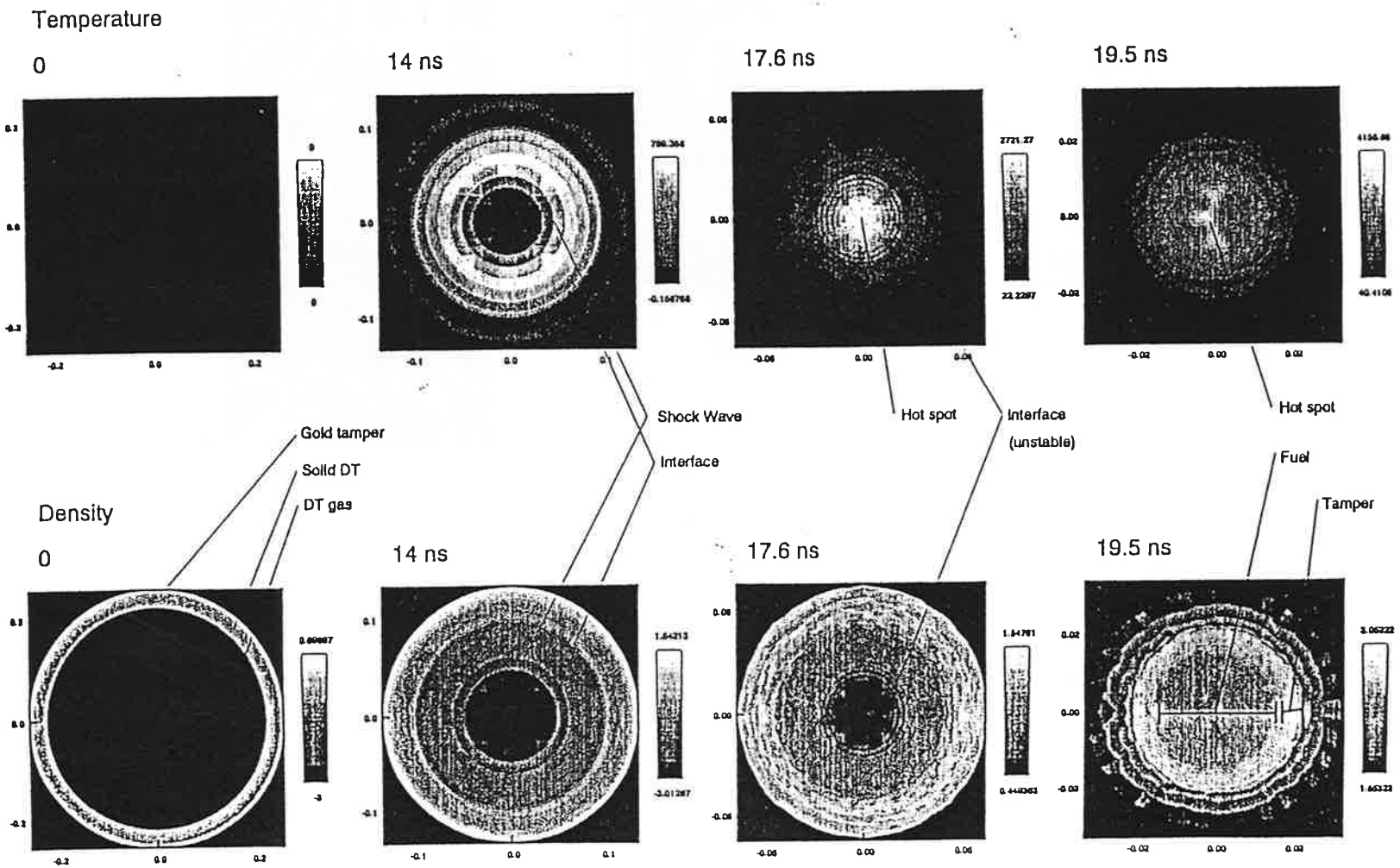

Linear grown of a Rayleigh-Taylor instability
Log(Xmax-Xmin)
0
-1
-2
-3
sqrt(kg)
Multi 2D
0
2e-07
4e-07
6e-07
8e-07
1e-06
time

Fig. 2

Simulation of a pressure-driven implosion



Gold foil irradiated by 100 eV thermal bath

Temperature plots

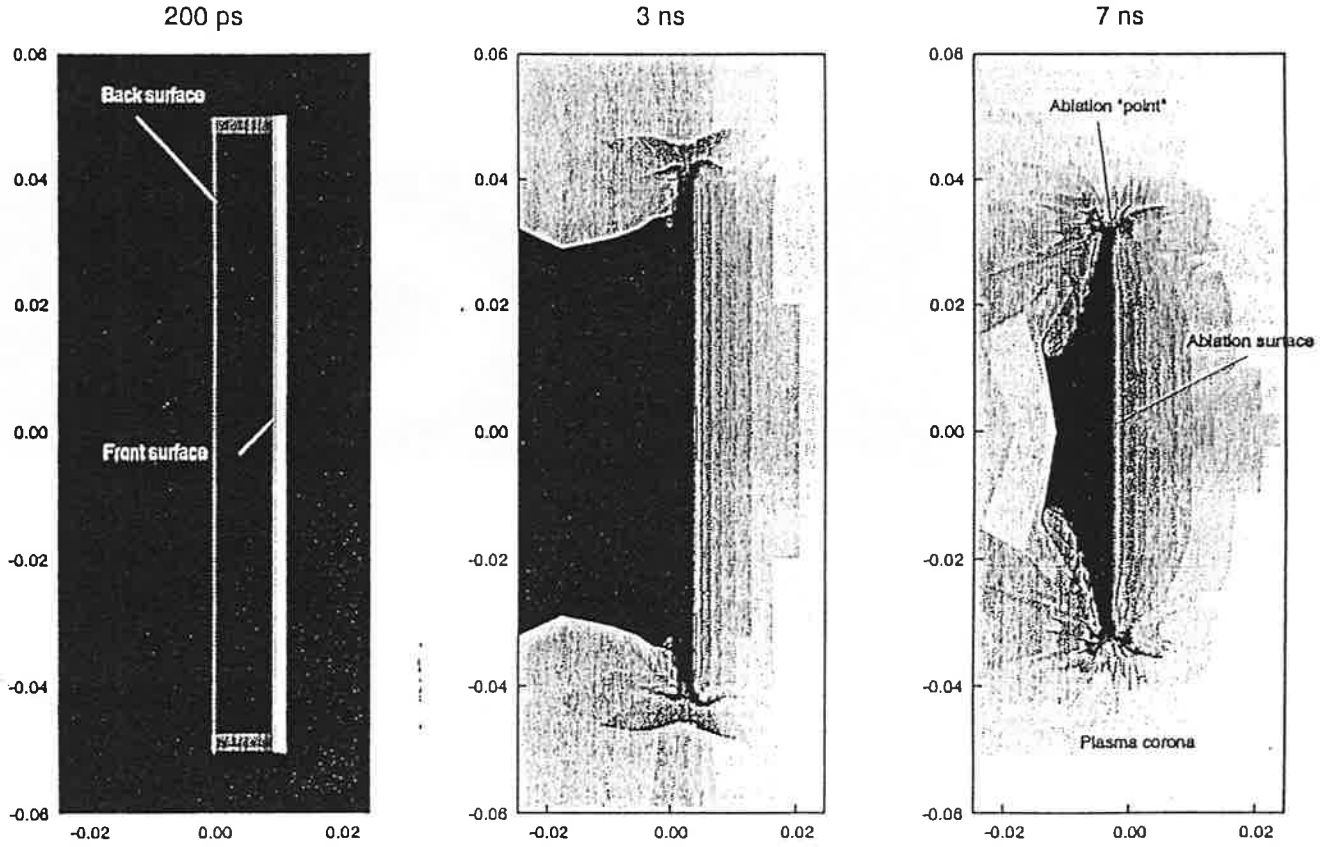
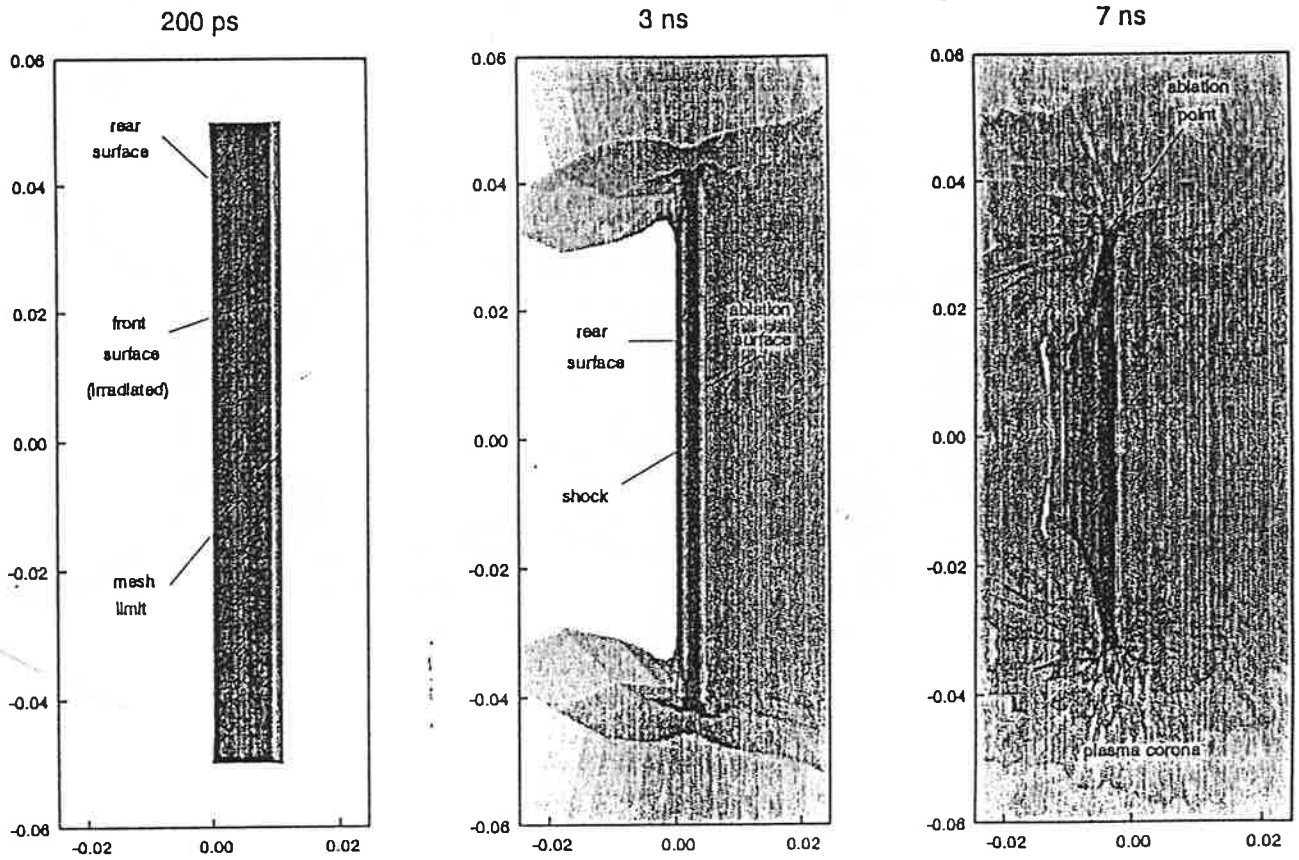
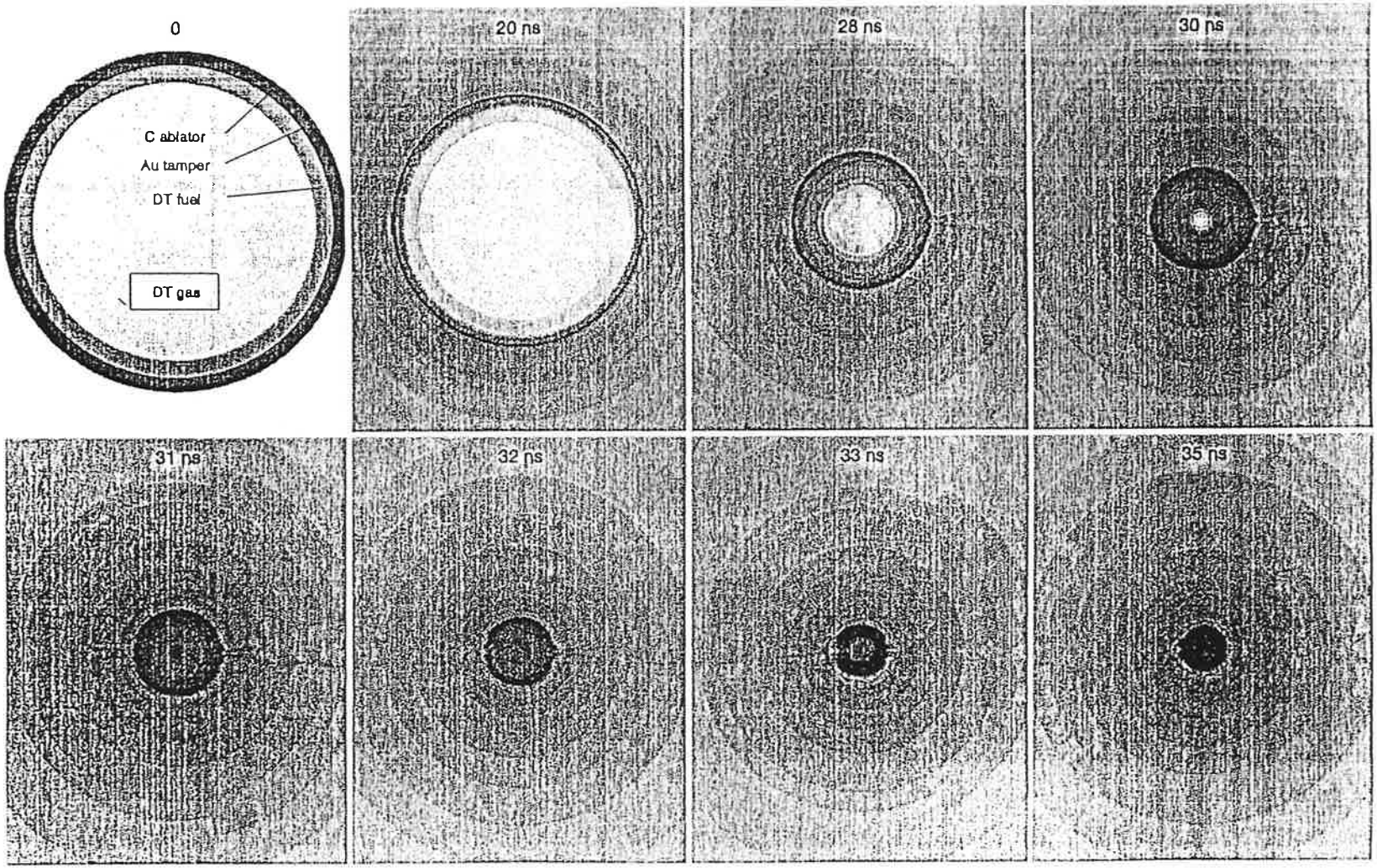


Fig 9

Gold foil irradiated by 100 eV thermal bath

Density plots





Density plots

fig. 12

fig. 11

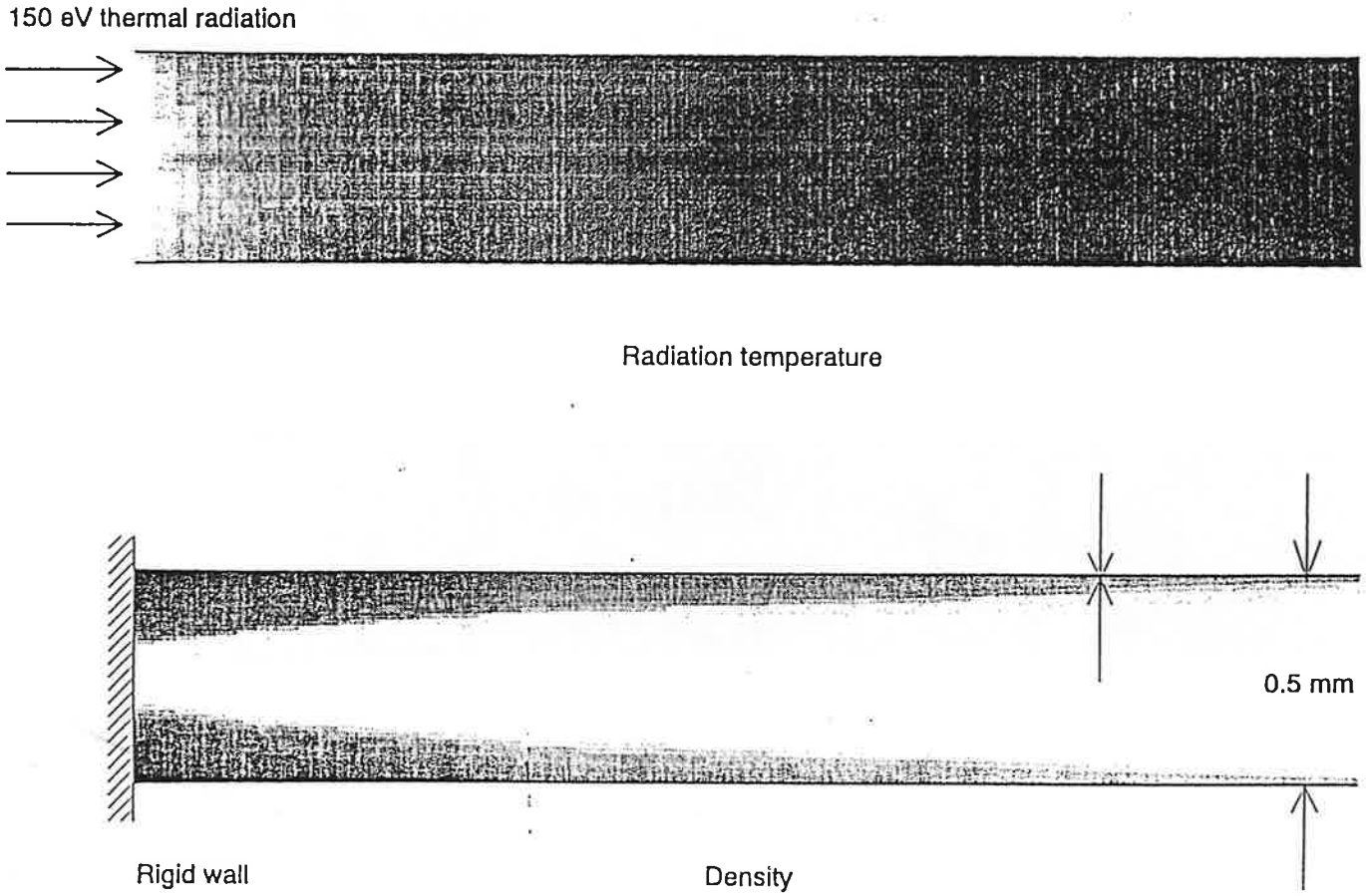
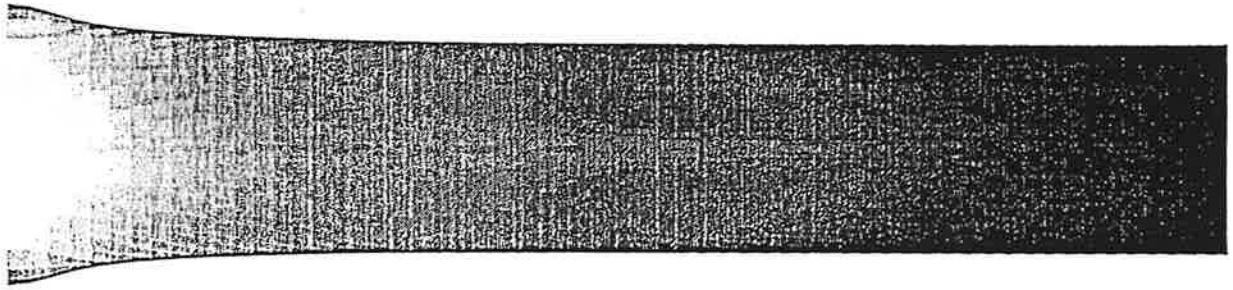
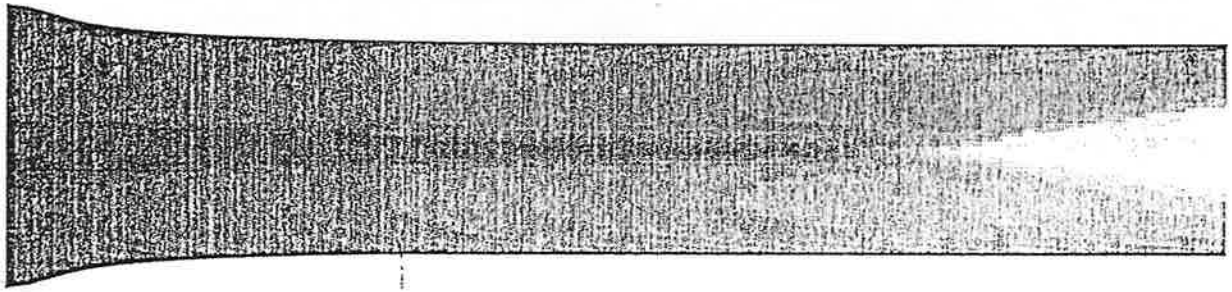

150 eV thermal radiation
Radiation temperature
0.5 mm
Rigid wall
Density

Fig. 12



Radiation temperature



Density

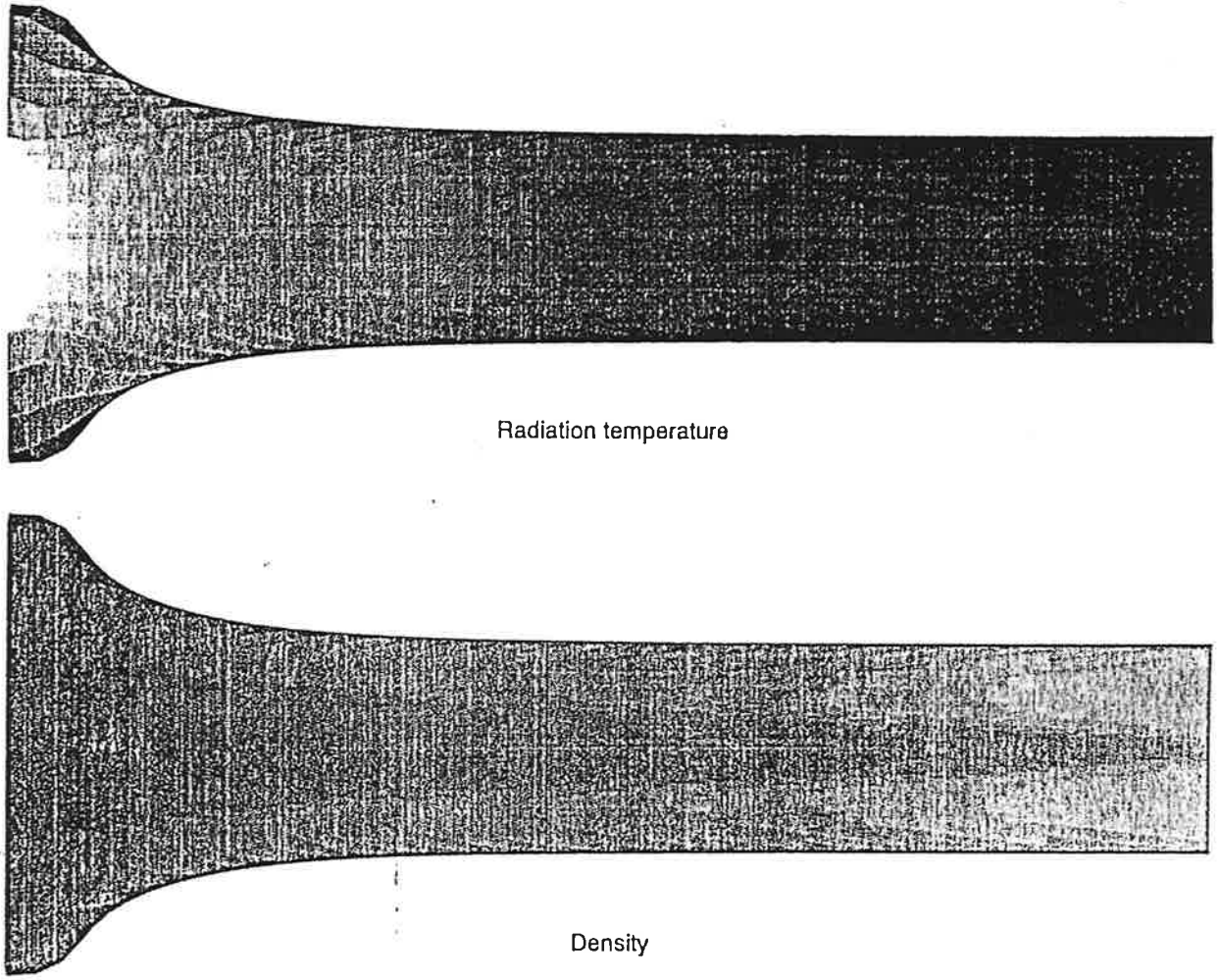
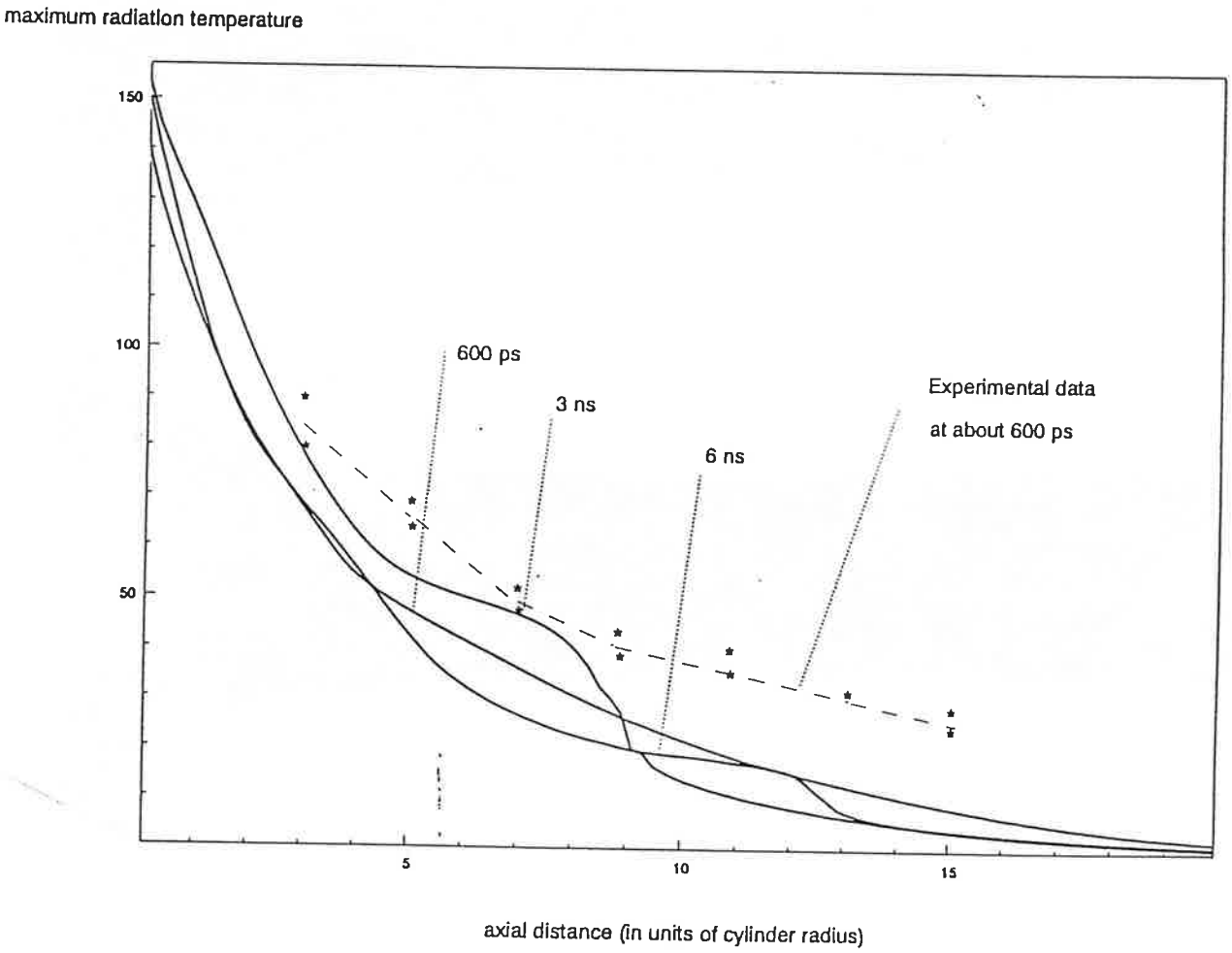


fig. 13

fig. 47


maximum radiation temperature
150
100
50
600 ps
3 ns
6 ns
Experimental data
at about 600 ps
5
10
15
axial distance (in units of cylinder radius)

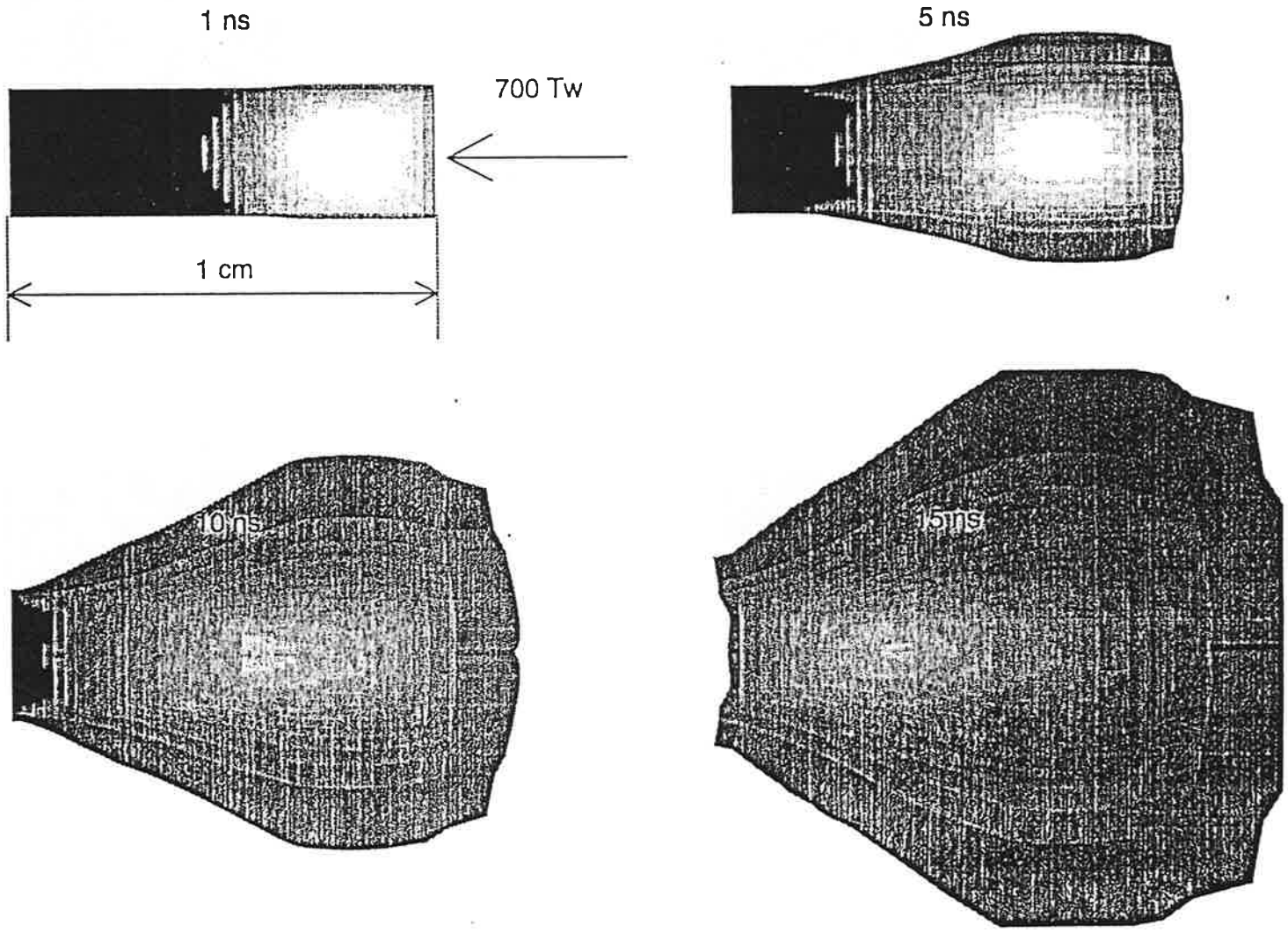
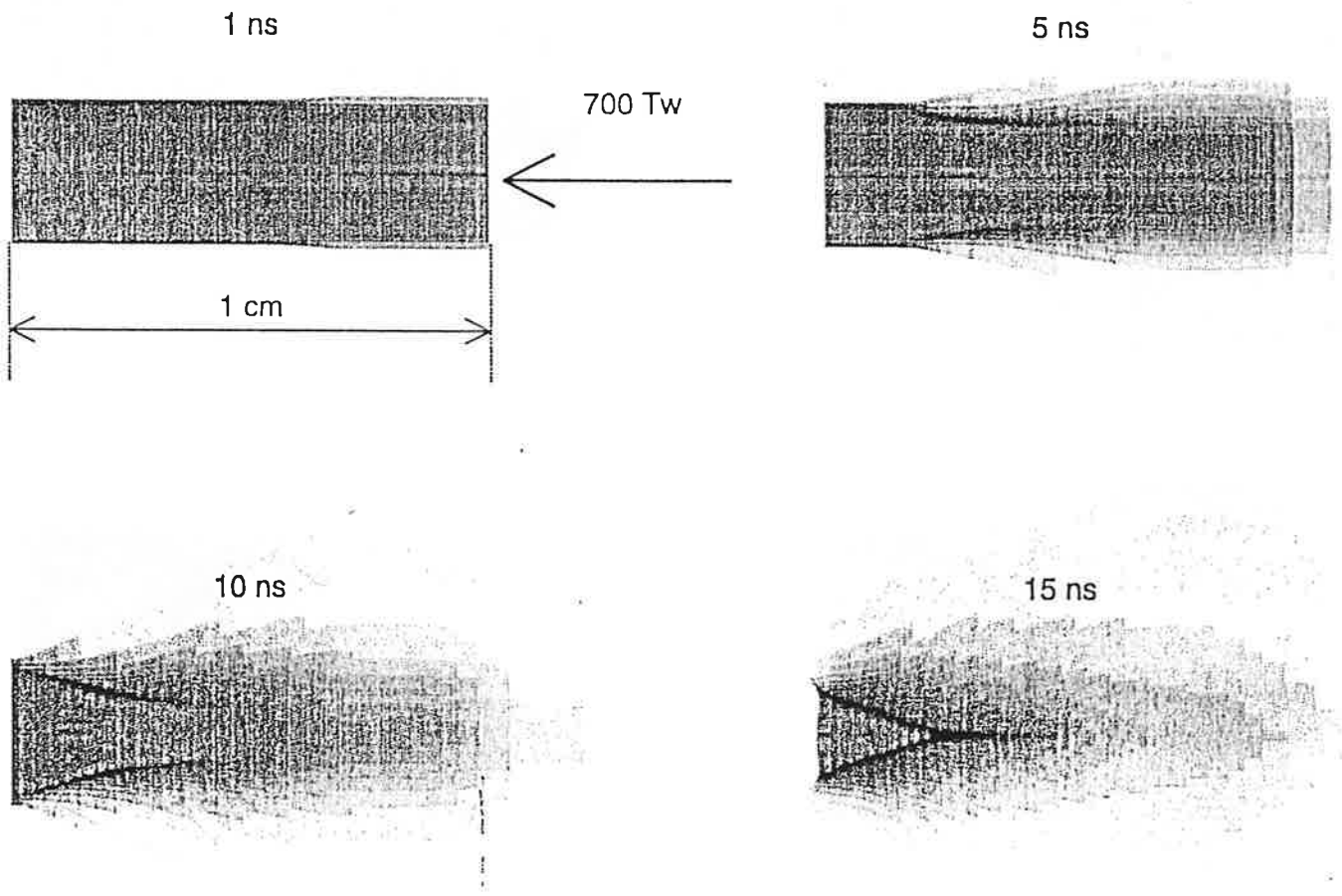
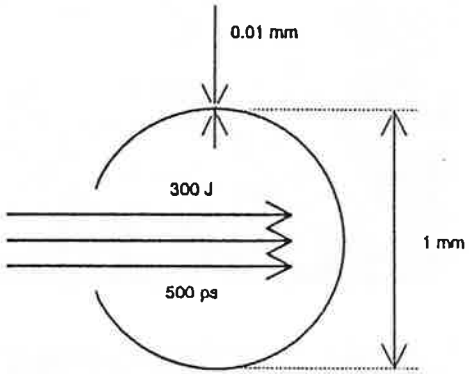
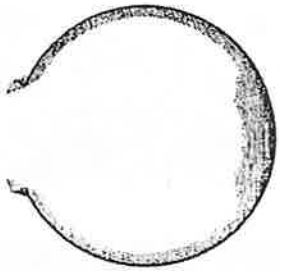


Fig. 15

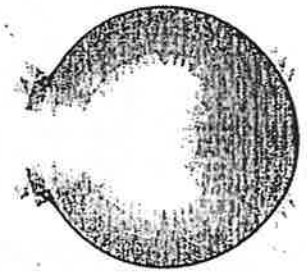

1 ns
700 Tw
5 ns
1 cm
10 ns
15 ns



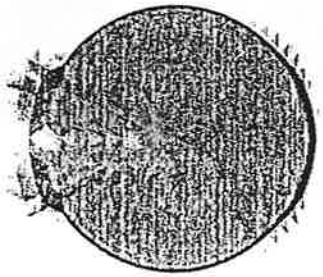
0



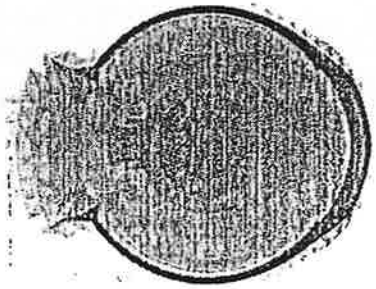
350 ps



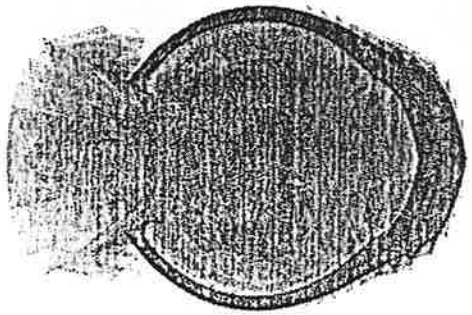
1 ns



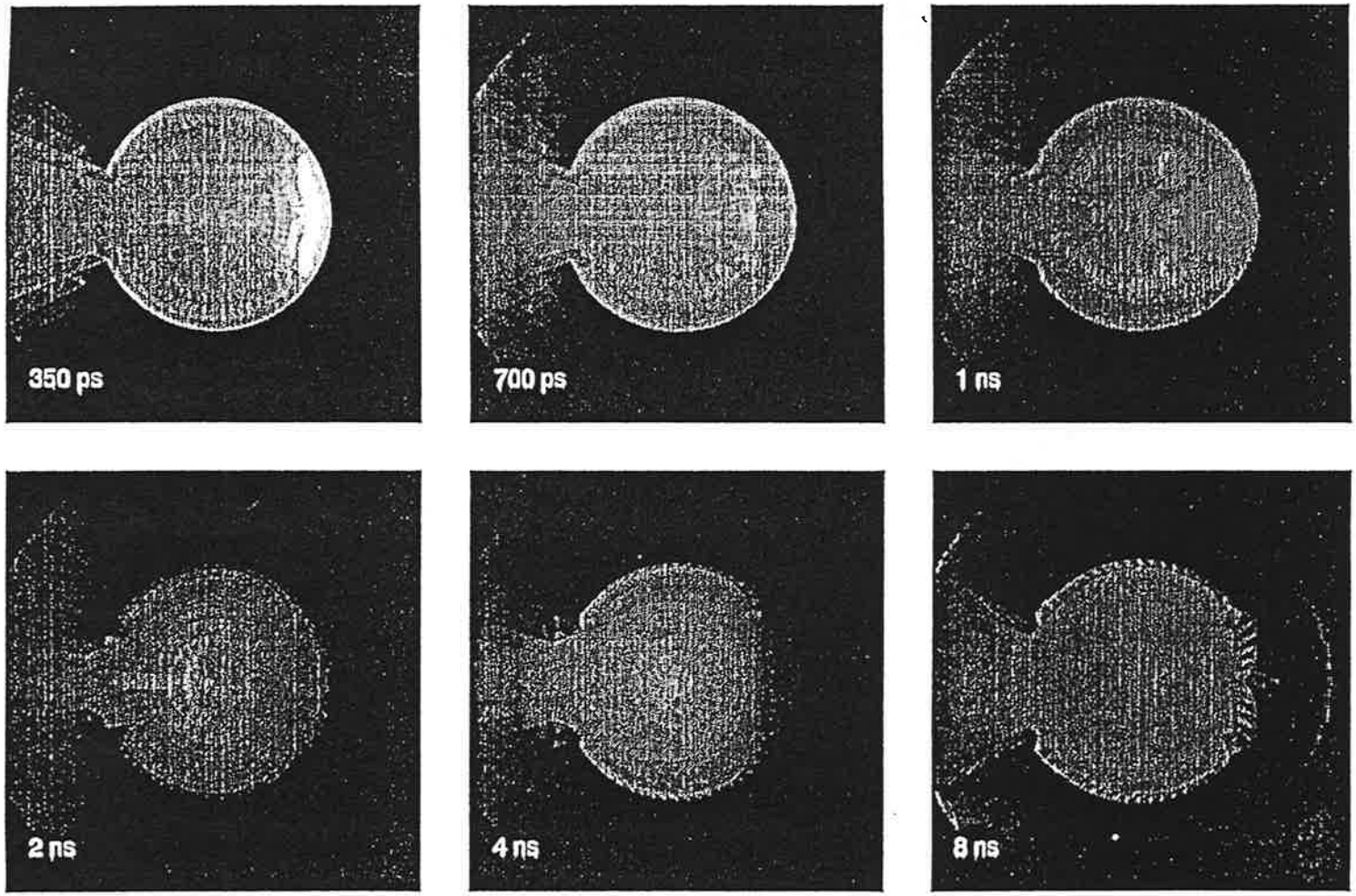
2 ns



4 ns



8 ns


350 ps
700 ps
1 ns
2 ns
4 ns
8 ns

Cavity simulation: implosion phase (pressure plots)

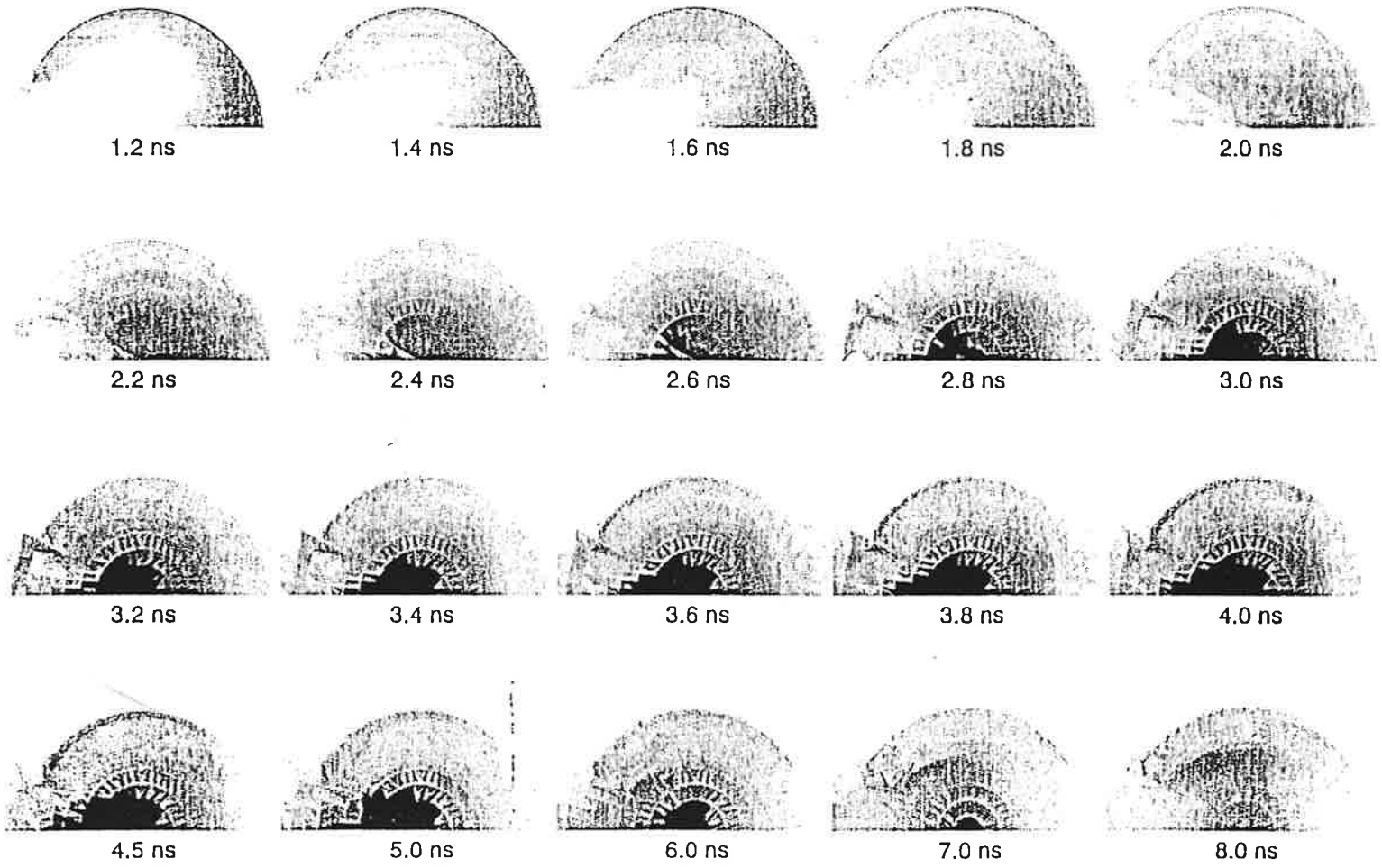


fig. 19

simulation of a laser-irradiated cavity

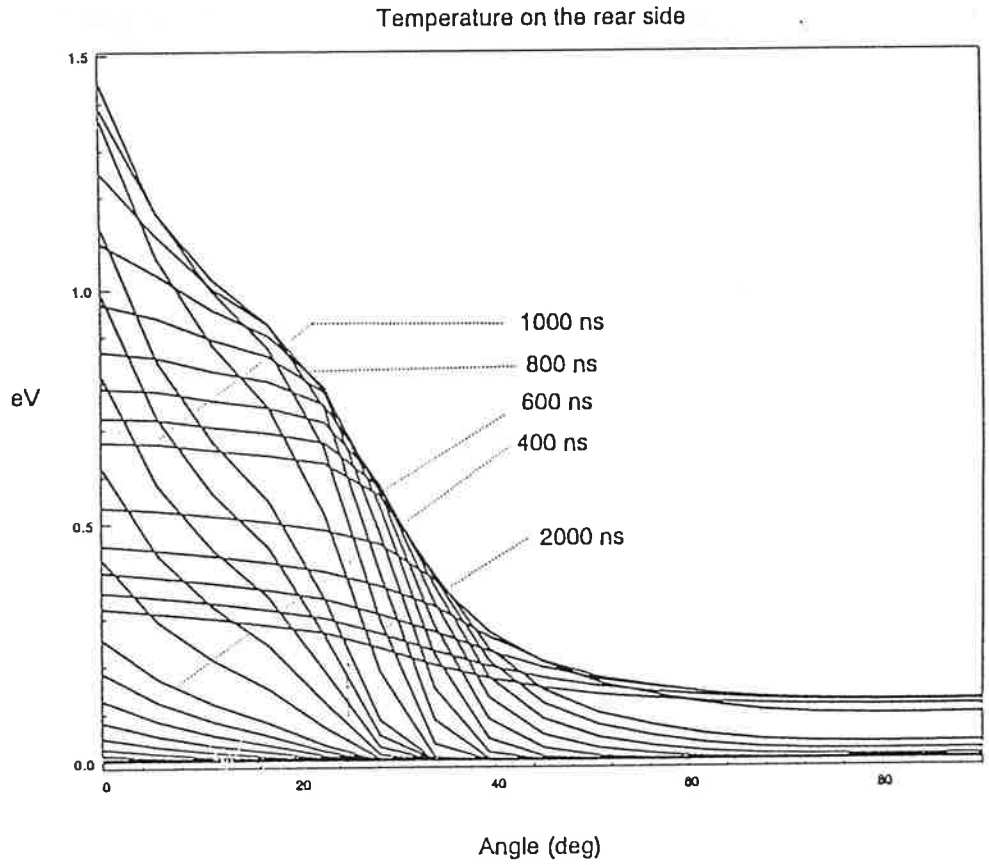


Fig. 10